

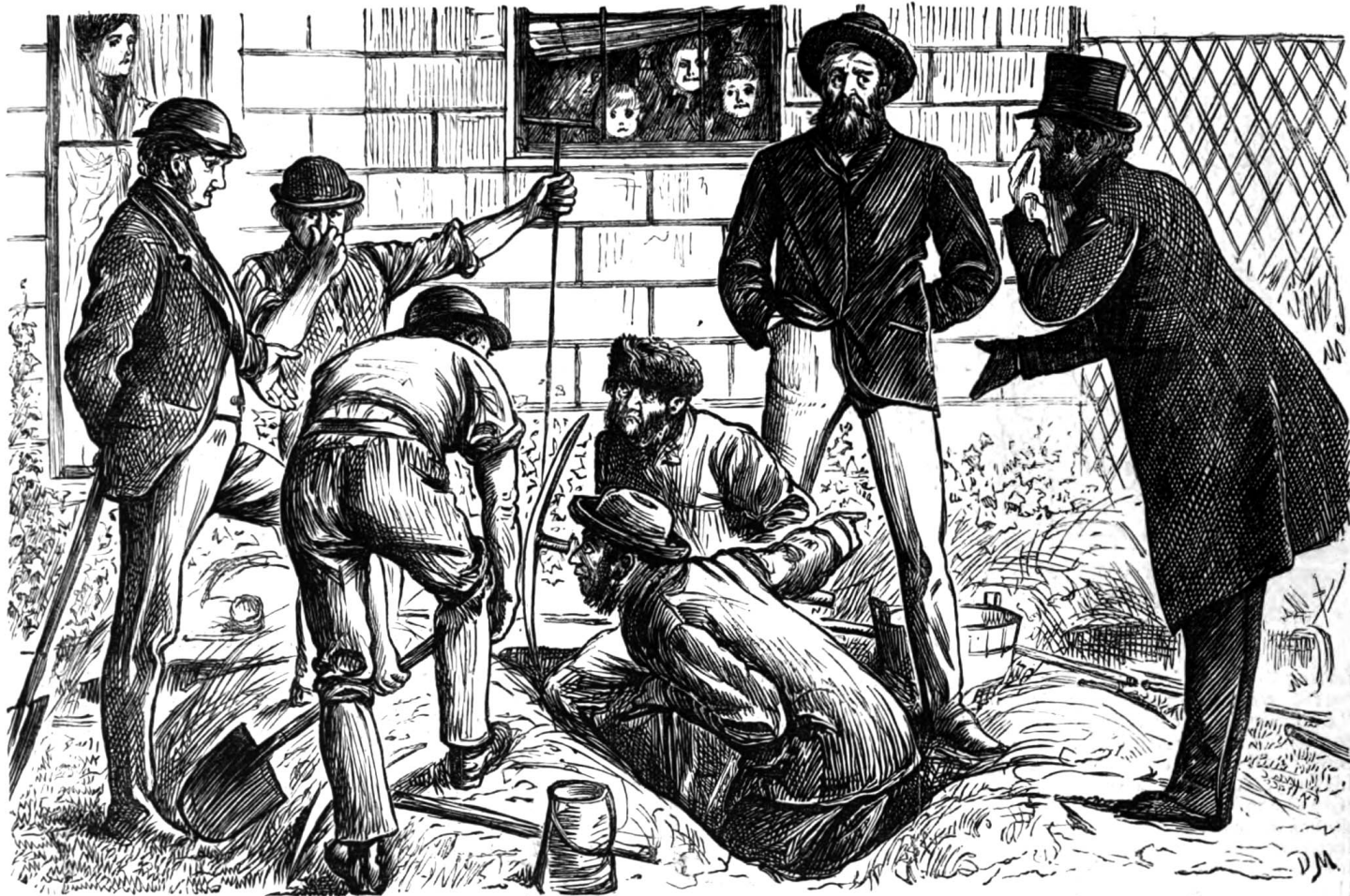


Beneath Apple DOS and Beneath Apple ProDOS 2020

By Don Worth and Pieter Lechner

Beneath Apple DOS

Beneath Apple ProDOS



Are you sure you want to go Beneath Apple DOS?

Original Copyright © 1981 by Quality Software.

Put into the Public Domain by Don Worth.

This edition edited and produced by Snape Publishing.

The word APPLE and the Apple logo are registered trademarks of Apple Computer, Inc. Apple Computer, Inc. was not in any way involved in the writing or other preparation of this manual, nor were the facts presented here reviewed for accuracy by that company. Use of the term APPLE should not be construed to represent any endorsement, official or otherwise, by Apple Computer, Inc.

Disclaimer: Quality Software and Snape Publishing shall have no liability or responsibility to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual or its use, including, but not limited to, any interruption in service, loss of business or anticipatory profits, or consequential damages resulting from the use of this product.

The body typeface is Garamond, while the header and subheads are Helvetica Neue LT Std. The program listings are in Courier 10 Pitch BT. The figure captions are Futura Std. The images are from the *Punch* series of political cartoons from the late 1800s. The figures and tables are drawn in Adobe Photoshop® and the book layout was prepared in Adobe InDesign®.



Contents

Beneath Apple DOS	1
Beneath Apple ProDOS	151
Miscellanea and Glossary.....	389



Early Disk Drive

From *Beneath Apple DOS*:

Thanks go to Vic Tolomei for his assistance in dissecting DOS 3.1 and to Lou Rivas for his patient proofreading. Thanks also to my wife Carley for putting up with the clackety clack of my Diablo long into the night.

Don D. Worth

Thanks to the people at Computerland of South Bay (California) who lent me support both of their time and equipment, and special thanks to John Gattuso, whose encouragement helped me to complete the task.

Pieter M. Lechner

From *Beneath Apple ProDOS*:

This book is dedicated to my sister, Betsy, who said she had room on her bookshelf for another one of my books.

Don D. Worth

This book is dedicated to my Father and Mother, with a deep sense of appreciation and gratitude.

Pieter M. Lechner

The authors wish to thank Quality Software for their able assistance in producing this book. Special thanks to Bob Christiansen, Bob Pierce, Kathy Schmidt, George Garcia, Vic Grenrock, and Jeff Weinstein for their unique and special contributions.

From the Editor:

This book combines two works from the 1980s: *Beneath Apple DOS* and *Beneath Apple ProDOS*. I have tried to keep as much of the original text as possible, but some of the figures and tables have been re-drawn to either make them easier to read or easier to understand (or both).

Also, some of the information is duplicated in both books, but I have kept the duplicates so if you're studying one operating system, you don't need to reference the other. The one main item I combined were the glossaries, since many of the same terms are in both. It is the last chapter in the book.

My personal history with *Beneath Apple DOS* goes back to high school. I remember someone brought the book into the computer room and we all marvelled at the detail and technical savvy in it. This was an explicit look at DOS that went far beyond anything before. I did go home and print out a disassembly of DOS 3.3 and go through each section, labeling the subroutines as listed in the book.

I still think reading code from others is one of the best ways to learn a programming language and I learned quite a bit from DOS 3.3. I learned enough that I eventually wrote my own DOS, using just the RWTs routines (I never did understand completely how it worked!) using my own routines for the rest of it. At the time, our school had a timesharing HP 2000F mainframe, and I used some of the syntax from that for my own DOS. Sadly, my version of DOS has been lost to history.

If this is your first time seeing this book, I hope you experience the joy so many have over the years. If you had this book previously, I hope I've kept the feel of the original book, while updating it for a new generation.

I'd like to specially thank Sean McNamara for his proofreading work on this book. Thank you, Sean!

— John Snape
Editor



Be careful using RWTS

Beneath Apple DOS

Contents

CHAPTER 1: INTRODUCTION	5
CHAPTER 2: THE EVOLUTION OF DOS	7
DOS 3 - 29 JUNE 1978	7
DOS 3.1 - 20 JULY 1978	7
DOS 3.2 - 16 FEBRUARY 1979	7
DOS 3.2.1 - 31 JULY 1979	8
DOS 3.3 - 25 AUGUST 1980	9
DOS 3.3 - 1 JANUARY 1983	10
CHAPTER 3: DISKETTE FORMATTING	11
TRACK FORMATTING	13
DATA FIELD ENCODING	21
SECTOR INTERLEAVING	25
CHAPTER 4: DISKETTE ORGANIZATION	31
DISKETTE SPACE ALLOCATION	32
HOW DOS IS STORED ON THE DISK	32
THE VTOC	33
VOLUME TABLE OF CONTENTS (VTOC) FORMAT	33
BIT MAPS OF FREE SECTORS ON A GIVEN TRACK	34
THE CATALOG	36
CATALOG SECTOR FORMAT	37
FILE DESCRIPTIVE ENTRY FORMAT	37
THE TRACK/SECTOR LIST	38
TRACK/SECTOR LIST FORMAT	39
TEXT FILES	42
BINARY FILES	43
APPLESOFT AND INTEGER FILES	43
OTHER FILE TYPES (S, R, NEW A, NEW B)	43
EMERGENCY REPAIRS	45

CHAPTER 5: THE STRUCTURE OF DOS	49
DOS MEMORY USE	49
THE DOS VECTORS IN PAGE 3	50
DOS VECTOR TABLE (\$3D0-\$3FF).....	51
WHAT HAPPENS DURING BOOTING.....	52
CHAPTER 6: USING DOS FROM ASSEMBLY LANGUAGE	57
CAVEAT	57
DIRECT USE OF DISK DRIVE.....	57
STEPPER PHASE OFF/ON.....	58
MOTOR OFF/ON.....	59
ENGAGE DRIVE 1/2	59
READ A BYTE	59
SENSE WRITE PROTECT	59
LOAD AND WRITE A BYTE	59
CALLING READ/WRITE TRACK/SECTOR (RWTS)	60
INPUT/OUTPUT CONTROL BLOCK - GENERAL FORMAT	60
RWTS IOB BY CALL TYPE	61
CALLING THE DOS FILE MANAGER	63
FILE MANAGER PARAMETER LIST - GENERAL FORMAT.....	65
FILE MANAGER PARAMETER LIST BY CALL TYPE.....	66
DOS BUFFERS	70
DOS BUFFER FORMAT	70
THE FILE MANAGER WORKAREA.....	70
FILE MANAGER WORKAREA FORMAT	71
COMMON ALGORITHMS.....	72
LOCATE A FREE DOS BUFFER	72
WHICH VERSION OF DOS IS ACTIVE?	72
IS DOS IN THE MACHINE?.....	73
WHICH BASIC IS SELECTED?	73
SEE IF A BASIC PROGRAM IS IN EXECUTION.....	73
PROCESS DOS ERROR FROM ASSEMBLY LANGUAGE.....	74
CHAPTER 7: CUSTOMIZING DOS	75
SLAVE VS MASTER PATCHING.....	75
AVOIDING RELOAD OF LANGUAGE CARD.....	76
INSERTING A PROGRAM BETWEEN DOS AND ITS BUFFERS	77
BRUN OR EXEC THE HELLO FILE.....	77
REMOVING THE PAUSE DURING A LONG CATALOG.....	78
WRITING DOS IN MEMORY TO A DISKETTE.....	78

CHAPTER 8: DOS PROGRAM LOGIC	79
DISK II CONTROLLER CARD ROM - BOOT 0.....	79
FIRST RAM BOOTSTRAP LOADER - BOOT 1	80
DOS 3.3 MAIN ROUTINES.....	81
DOS ZERO PAGE USAGE	115
APPENDIX A: EXAMPLE PROGRAMS	117
STORING THE PROGRAMS ON DISKETTE	118
DUMP – TRACK DUMP UTILITY	120
ZAP – DISK UPDATE UTILITY	124
INIT – REFORMAT A SINGLE TRACK	128
FTS – FIND T/S LISTS UTILITY.....	132
COPY – CONVERT FILES.....	137
APPENDIX B: DISK PROTECTION SCHEMES	145



It says here that file
starts on Track \$18,
Sector \$ØF

Chapter 1

Introduction

Beneath Apple DOS is intended to serve as a companion to Apple's *DOS Manual*, providing additional information for the advanced programmer or the novice Apple user who wants to know more about the structure of diskettes. It is not the intent of this manual to replace the documentation provided by Apple Computer Inc. Although, for the sake of continuity, some of the material covered in the Apple manual is also covered here, it will be assumed that the reader is reasonably familiar with the contents of the *DOS Manual*. Since all chapters presented here may not be of use to each Apple owner, each has been written to stand on its own.

The information presented here is a result of intensive disassembly and annotation of various versions of DOS by the authors and by other experienced systems programmers. It also draws from application notes, articles, and discussions with knowledgeable people. This manual was not prepared with the assistance of Apple Computer Inc. Although no guarantee can be made concerning the accuracy of the information presented here, all of the material included in *Beneath Apple DOS* has been thoroughly researched and tested.

There were several reasons for writing *Beneath Apple DOS*:

- To show direct assembly language access to DOS.
- To help you to fix clobbered diskettes.
- To correct errors and omissions in the Apple documentation.
- To allow you to customize DOS to fit your needs.
- To provide complete information on diskette formatting

When Apple Computer Inc. introduced its Disk Operating System (DOS) version 3 in 1978 to support the new Disk II drive, very little documentation was provided. Later, when DOS 3.2 was released, a 178-page instructional and reference manual became available covering the use of DOS from BASIC in depth and even touched upon some of the internal workings of DOS. With the advent of DOS 3.3, the old 3.2 manual was updated but the body of information in it remained essentially intact. Beyond these Apple manuals,

there have been no significant additions to the documentation on DOS, apart from a few articles in Apple user group magazines and newsletters. This manual takes up where the *Disk Operating System Manual* leaves off.

Throughout this manual, discussion centers primarily on DOS version 3.3. The reasons for this are that 3.3 was the most recent release of DOS at the time of this writing and that it differs less from DOS 3.2 than one would imagine. Wherever there is a major difference between the various DOS releases in a given topic, each release will be covered.

In addition to the DOS dependent information provided, many of the discussions also apply to other operating systems on the Apple II and Apple III. For example, disk formatting at the track and sector level is, for the most part, the same.



one week



two years

Programmer Age Progression

Chapter 2

The Evolution of DOS

Since its introduction, Apple DOS has gone through three major versions. All of these versions look very much the same on the surface. All commands supported by DOS 3.3 are also supported in 3.2 and 3.1. The need for additional versions has been more to fix errors in DOS and to make minor enhancements than to provide additional functionality. Only DOS 3.3 has offered any major improvement in function; an increase in the number of sectors that will fit on a track from 13 to 16.

DOS 3 - 29 JUNE 1978

DOS 3.1 - 20 JULY 1978

The first release of DOS was apparently a victim of a rush at Apple to introduce the Disk II. As such, it had a number of bugs. With the movement towards the Apple II Plus and the introduction of the Autostart ROM, a new release was needed.

DOS 3.2 - 16 FEBRUARY 1979

Although DOS 3.2 embodied more changes from its predecessor than any other release of DOS, 90% of the basic structure of DOS 3.1 was retained. The major differences between DOS 3.1 and 3.2 and later versions of DOS are listed below:

- **NOMON C, I, O** is the initial default under DOS 3.2. **MON C, I, O** was the default under DOS 3.1.
- Input prompts (> , I , *) are echoed when **MON O** is in effect, not under **MON I** as was the case under 3.1.
- When a DOS command was entered from the keyboard, DOS executed it and then passed a blank followed by a carriage return to BASIC under 3.1. Under 3.2 only a carriage return is passed.

- Under 3.2, certain commands may not be entered from the keyboard but may only be used within a BASIC program (**READ**, **WRITE**, **POSITION**, **OPEN**, **APPEND**).
- Under 3.2, when **LOAD**ing an Applesoft program, DOS automatically converts from Applesoft ROM format to Applesoft RAM format if the RAM version of BASIC is in use and vice versa.
- DOS 3.1 could not read lower case characters from a text file; DOS 3.2 can.
- Some DOS commands are allowed to create a new file, others will not. Under DOS 3.1, any reference to a file that didn't exist, caused it to be created. This forced DOS 3.1 to then delete it if a new file was not desired. (**LOAD XYZ** under 3.1 if **XYZ** did not exist, created **XYZ**, deleted **XYZ**, and then printed the file not found error message.) Under 3.2, **OPEN** is allowed to create a file if one does not exist, but **LOAD** may not.
- Under 3.1, exiting to the monitor required that the monitor status register location (\$48) be set to zero before reentering DOS. Under DOS 3.2 this is no longer necessary.
- The Read/Write-Track/Sector (RWTS) section of DOS disables interrupts while it is executing. Under 3.1, RWTS could be interrupted by a peripheral while writing to a disk, destroying the disk.
- The default for the B (byte offset) keyword is 0 under 3.2.
- DOS was reassembled for 3.2 causing most of its interesting locations and routines to move slightly. This played havoc with user programs and utilities which had DOS addresses built into them.
- Additional file types (beyond T, I, A, and B) are defined within DOS 3.2, although no commands yet support them. The new types are S, R, a new A, and a new B. R has subsequently been used by the DOS Toolkit for relocatable object module assembler files. At present, no other use is made of these extra file types.
- Support was added under 3.2 for the Autostart ROM.
- All files open when a disk full condition occurs are closed by DOS 3.2.
- As with each new release of DOS, several new programs were added to the master diskette for 3.2. Among these was **UPDATE 3.2**, a replacement for **MASTER CREATE**, the utility for creating master diskettes. **UPDATE 3.2** converts a slave into a master and allows the **HELLO** file to be renamed.

DOS 3.2.1 - 31 JULY 1979

DOS 3.2.1 was essentially a “maintenance release” of DOS 3.2. Minor patches were made to RWTS and the **COPY** program to correct a timing problem when a dual drive copy was done. Additional delays were added following a switch between drives.

DOS 3.3 - 25 AUGUST 1980

Introduced in mid 1980 as a hardware/software upgrade from DOS 3.2.1, the DOS 3.3 package includes new bootstrap and state ROM chips for the disk controller card which provide the capability to format, read, and write a diskette with 16 sectors. (These ROMs are the same ones used with the Language System.) This improvement represents almost a 25% increase in available disk space over the old 13 sector format. Also included in the 3.3 package is an updated version of the DOS manual, a BASICS diskette (for 13 sector boots), and a master diskette. Although the RWTS portion of DOS was almost totally rewritten, the rest of DOS was not reassembled and only received a few patches:

- The initial DOS bootstrap loader was moved to \$800 under 3.3. It was at \$300 under 3.2. In addition, as stored on the diskette (track 0 sector 0) it is nibbilized in the same way as all other sectors under 3.3.
- A bug in **APPEND** which caused it to position improperly if the file was a multiple of 256 bytes long was fixed under 3.3.
- A **VERIFY** command is internally executed after every **SAVE** or **BSAVE** under 3.3.
- All 4 bytes are used in the Volume Table Of Contents (VTOC) free sector bit map when keeping track of free sectors. This allows DOS to handle up to 32 sectors per track. Of course, RWTS will only handle 16 sectors due to hardware limitations.
- If a Language Card is present, DOS stores a zero on it at \$E000 during bootstrap to force the **HELLO** program on the master diskette to reload BASIC.
- DOS is read into memory from the top down (backwards) under 3.3 rather than the bottom up. Its image is still stored in the same order on the diskette (tracks 0, 1, and 2), however.
- Additional programs added to the master diskette under 3.3 include **FID**, a generalized file utility which allows individual files or groups of files

to be copied, **MUFFIN**, a conversion copy routine to allow 3.2 files to be moved to 16 sector 3.3 diskettes, **BOOT 13**, a program which will boot a 13 sector diskette, and a new **COPY** program which will also support single drive copies.

- Under 3.2, speed differences in some drives prevented their use together with the DOS **COPY** program. Because the **COPY** program was rewritten under 3.3, that restriction no longer applies.

DOS 3.3 - 1 JANUARY 1983

This “maintenance release” of DOS was introduced with the Apple IIe computer. It contains a few minor patches and no additional function.

- A patch was introduced in DOS 3.3 to fix a bug in **APPEND** processing. This patch also had bugs. Additional patches were added to (hopefully) correct this problem.
- An error in the **POSITION** calculation for large files is corrected in this release.
- A few instructions were added to properly support the 80 column display on the Apple IIe.
- The System Master diskette contains some different files. Notably, a fast loader for the language card and all of the example programs have been moved to a separate diskette.

Chapter 3

Diskette Formatting

Apple Computer's excellent manual on the Disk Operating System (DOS) provides only very basic information about how diskettes are formatted. This chapter will explain in detail how information is structured on a diskette. The first section will contain a brief introduction to the hardware, and may be skipped by those already familiar with the DOS manual.

For system housekeeping, DOS divides diskettes into tracks and sectors. This is done during the initialization process. A track is a physically-defined circular path which is concentric with the hole in the center of the diskette. Each track is identified by its distance from the center of the disk. Similar to a phonograph stylus, the read/write head of the disk drive may be positioned over any given track. The tracks are similar to the grooves in a record, but they are not connected in a spiral. Much like playing a record, the diskette is spun at a constant speed while the data is read from or written to its surface with the read/write head. Apple formats its diskettes into 35 tracks. They are numbered from 0 to 34, track 0 being the outermost track and track 34 the innermost. Figure 3.1 illustrates the concept of tracks, although they are invisible to the eye on a real diskette.

It should be pointed out, for the sake of accuracy, that the disk arm can position itself over 70 "phases". To move the arm past one track to the next,

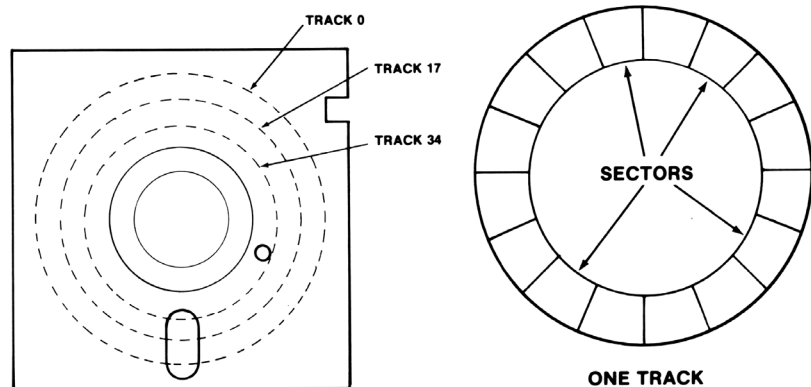


FIGURE 3.1

Figure 3.1 Tracks and Sectors

two phases of the stepper motor, which moves the arm, must be cycled. This implies that data might be stored on 70 tracks, rather than 35. Unfortunately, the resolution of the read/write head and the accuracy of the stepper motor are such, that attempts to use these phantom “half” tracks create so much cross-talk that data is lost or overwritten. Although the standard DOS uses only even phases, some protected disks use odd phases or combinations of the two, provided that no two tracks are closer than two phases from one another. See Appendix B for more information on protection schemes.

A sector is a subdivision of a track. It is the smallest unit of “updatable” data on the diskette. DOS generally reads or writes data a sector at a time. This is to avoid using a large chunk of memory as a buffer to read or write an entire track. Apple DOS has used two different track formats—one divides the track into 13 sectors, the other, into 16 sectors. The sectoring does not use the index hole, provided on most diskettes, to locate the first sector of the track. The implication is that the software must be able to locate any given track and sector with no help from the hardware. This scheme, known as “soft sectoring”, takes a little more space for storage but allows flexibility, as evidenced by the recent change from 13 sectors to the present 16 sectors per track. The following table categorizes the amount of data stored on a diskette under both 13 and 16 sector formats.

DISK ORGANIZATION

TRACKS	
All DOS versions.....	35
SECTORS PER TRACK	
DOS 3.2.1 and earlier.....	13
DOS 3.3.....	16
SECTORS PER DISKETTE	
DOS 3.2.1 and earlier.....	455
DOS 3.3.....	560
BYTES PER SECTOR	
All DOS versions.....	256
BYTES PER DISKETTE	
DOS 3.2.1 and earlier.....	116480
DOS 3.3.....	143360
USABLE* SECTORS FOR DATA STORAGE	
DOS 3.2.1 and earlier.....	403
DOS 3.3.....	496
USABLE* BYTES PER DISKETTE	
DOS 3.2.1 and earlier.....	103168
DOS 3.3.....	126976

* Excludes DOS, VTOC, and CATALOG

TRACK FORMATTING

Up to this point we have broken down the structure of data to the track and sector level. To better understand how data is stored and retrieved, we will start at the bottom and work up.

As this manual is primarily concerned with software, no attempt will be made to deal with the specifics of the hardware. For example, while in fact data is stored as a continuous stream of analog signals, we will deal with discrete digital data, i.e., a 0 or a 1. We recognize that the hardware converts analog data to digital data but how this is accomplished is beyond the scope of this manual.

Data bits are recorded on the diskette in precise intervals. The hardware recognizes each of these intervals as either a 0 or a 1. We will define these intervals to be “bit cells”. A bit cell can be thought of as the distance the diskette moves in four machine cycles, which is about four microseconds. Using this representation, data written to and read back from the diskette takes the form shown in Figure 3.2. The data pattern shown represents a binary value of 101.

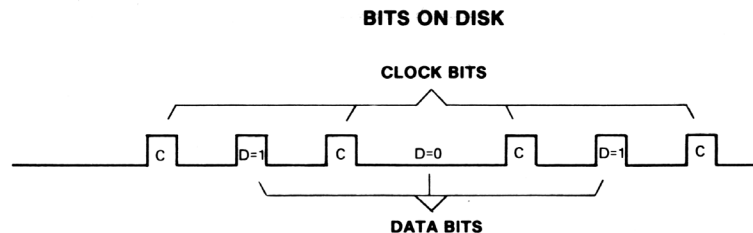


Figure 3.2 Bits on Diskette

A byte as recorded on the disk consists of eight (8) consecutive bit cells. The most significant bit cell is usually referred to as bit cell 7 and the least significant bit cell would be bit cell 0. When reference is made to a specific data bit (i.e. data bit 5), it is with respect to the corresponding bit cell (bit cell 5). Data is written and read serially, one bit at a time. Thus, during a write operation, bit cell 7 of each byte would be written first, with bit cell 0 being written last. Correspondingly, when data is being read back from the diskette, bit cell 7

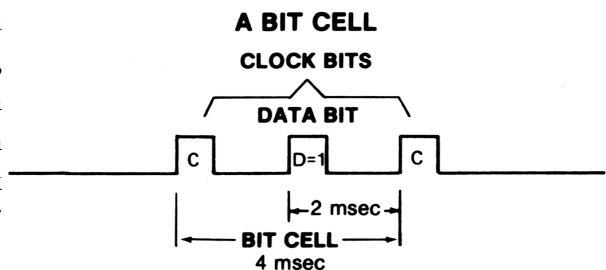


Figure 3.3 A Bit Cell

is read first and bit cell 0 is read last. Figure 3.4 illustrates the relationship of the bits within a byte.

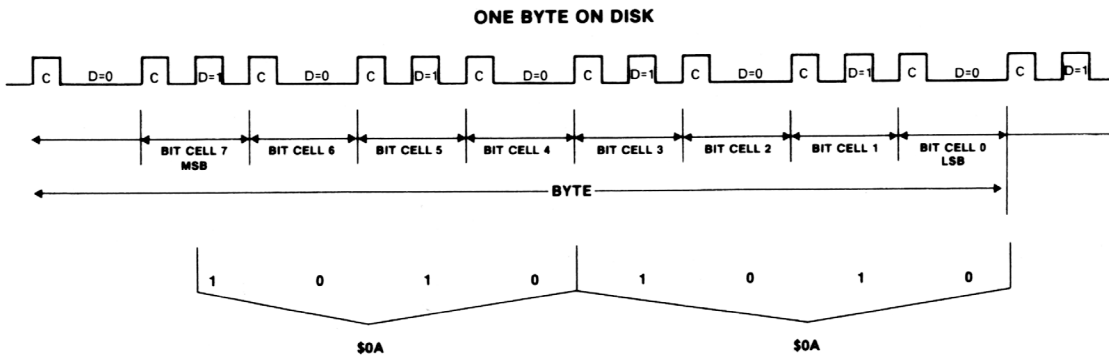


Figure 3.4 One Byte on Diskette

To graphically show how bits are stored and retrieved, we must take certain liberties. The diagrams are a representation of what functionally occurs within the disk drive. For the purposes of our presentation, the hardware interface to the diskette will be represented as an eight bit “data latch”. While the hardware involves considerably more complication, from a software standpoint it is reasonable to use the data latch, as it accurately embodies the function of data flow to and from the diskette.

Figure 3.5 shows the three bits, 101, being read from the diskette data stream into the data latch. Of course another five bits would be read to fill the latch. As can be seen, the data is separated from the clock bits. This task is done by the hardware and is shown more for accuracy than for its importance to our discussion.

Writing data can be depicted in much the same way (see Figure 3.6). The clock bits which were separated from the data must now be interleaved with the data as it is written. It should be noted that, while in write mode, zeros are being brought into the data latch to replace the data being written. It is the task of the software to make sure that the latch is loaded and instructed to write in 32-cycle intervals. If not, zero bits will continue to be written every four cycles, which is, in fact, exactly how self-sync bytes are created. Self-sync bytes will be covered in detail shortly.

A “field” is made up of a group of consecutive bytes. The number of bytes varies, depending upon the nature of the field. The two types of fields present on a diskette are the Address Field and the Data Field. They are similar in that they both contain a prologue, a data area, a checksum, and an epilogue. Each field on a track is separated from adjacent fields by a number of bytes. These

READING DATA FROM DISKETTE

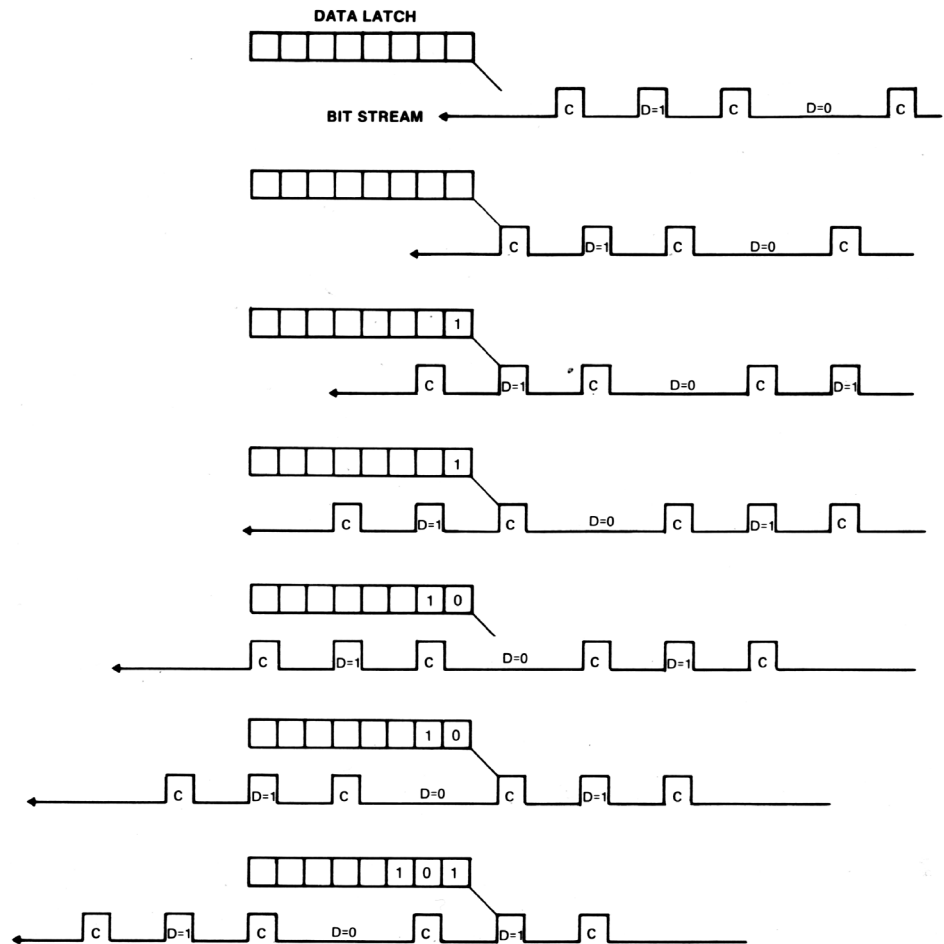


Figure 3.5 Reading Data from Diskette

areas of separation are called “gaps” and are provided for two reasons. One, they allow the updating of one field without affecting adjacent fields (on the Apple, only data fields are updated). Secondly, they allow the computer time to decode the address field before the corresponding data field can pass beneath the read/write head.

All gaps are primarily alike in content, consisting of self-sync hexadecimal FFs, and vary only in the number of bytes they contain. Figure 3.7 is a diagram of a portion of a typical track, broken into its major components.

Self-sync or auto-sync bytes are special bytes that make up the three different types of gaps on a track. They are so named because of their ability to automatically bring the hardware into synchronization with data bytes on

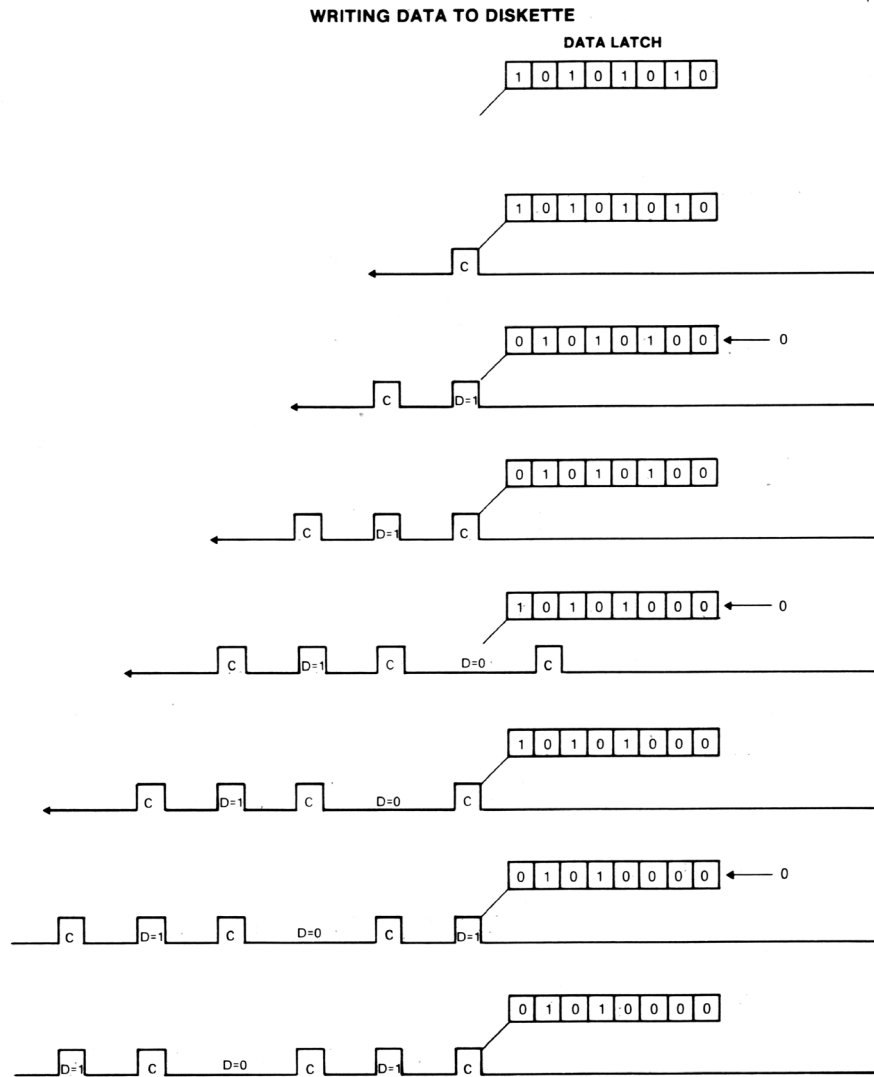
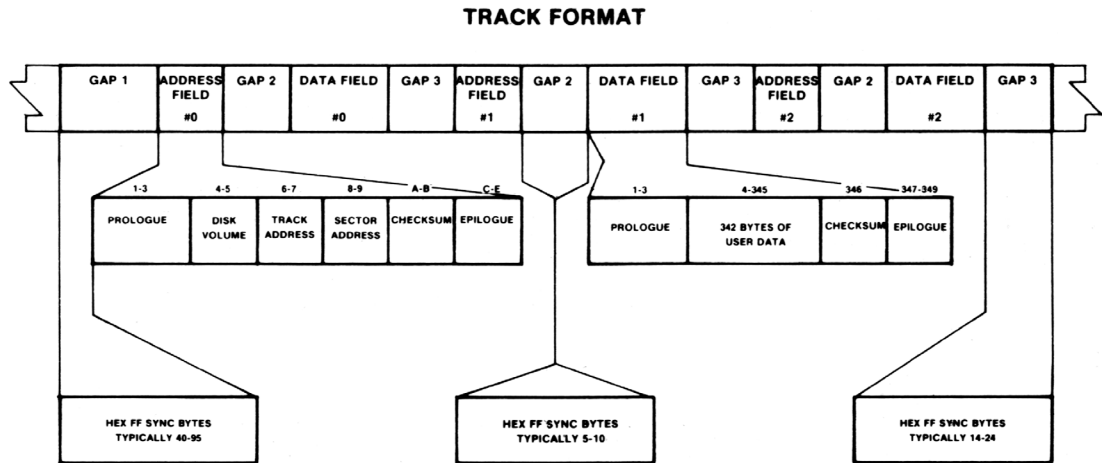


FIGURE 3.6

Figure 3.6 Writing Data to Diskette

the disk. The difficulty in doing this lies in the fact that the hardware reads bits and the data must be stored as eight bit bytes. It has been mentioned that a track is literally a continuous stream of data bits. In fact, at the bit level, there is no way to determine where a byte starts or ends, because each bit cell is exactly the same, written in precise intervals with its neighbors. When the drive is instructed to read data, it will start wherever it happens to be on a particular track. That could be anywhere among the 50,000 or so bits on a track. Distinguishing clock bits from data bits, the hardware finds the first bit

**Figure 3.7 Track Format**

cell with data in it and proceeds to read the following seven data bits into the eight bit latch. In effect, it assumes that it had started at the beginning of a data byte. Of course, in reality, the odds of its having started at the beginning of a byte are only one in eight. Pictured in Figure 3.8 is a small portion of a track. The clock bits have been stripped out and 0's and 1's have been used for clarity.

There is no way from looking at the data to tell what bytes are represented,

AN EXAMPLE BIT STREAM ON THE DISK

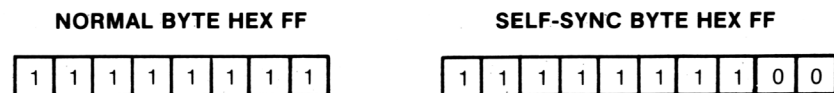
0 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 0 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1

Figure 3.8 Example Bit Stream

because we don't know where to start. This is exactly the problem that self-sync bytes overcome.

A self-sync byte is defined to be a hexadecimal **FF** with a special difference. It is, in fact, a 10 bit byte rather than an eight bit byte. Its two extra bits are zeros. Figure 3.9 shows the difference between a normal data hex **FF** that might be found elsewhere on the disk and a self-sync hex **FF** byte.

A self-sync is generated by using a 40 cycle (micro-second) loop while writing an **FF**. A bit is written every four cycles, so two of the zero bits brought into the

**Figure 3.9 Normal Byte and Self-Sync Byte**

data latch while the **FF** was being written are also written to the disk, making the 10 bit byte. (DOS 3.2.1 and earlier versions use a nine bit byte due to the

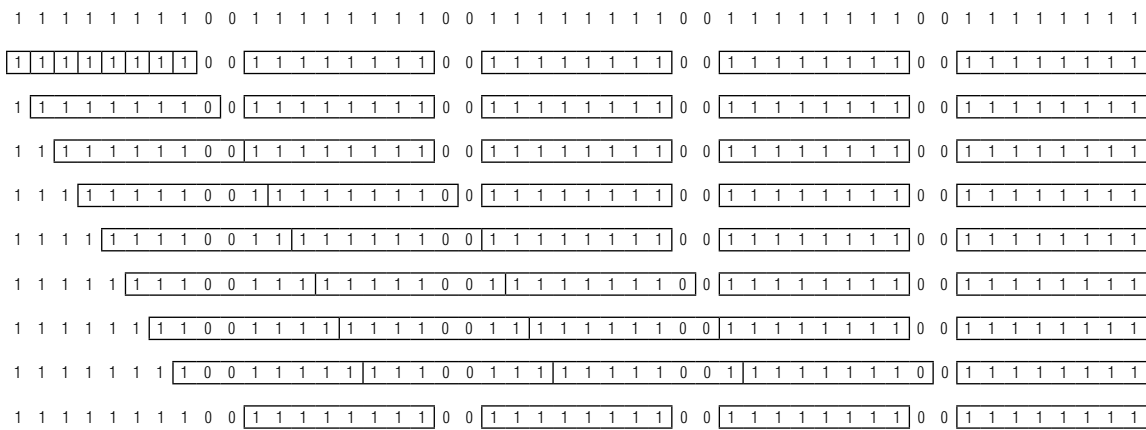


Figure 3.10 Five Auto-Sync Bytes

hardware's inability to always detect two consecutive zero bits.) It can be shown, using Figure 3.10, that five self-sync bytes are sufficient to guarantee that the hardware is reading valid data. The reason for this is that the hardware requires the first bit of a byte to be a 1. Pictured at the top of the figure is a stream of five FFs, four self-sync FFs followed by a normal FF. Each row below that demonstrates what the hardware will read should it start reading at any given bit in the first byte. In each case, by the time the five sync bytes have passed beneath the read/write head, the hardware will be "synced" to read the data bytes that follow. As long as the disk is left in read mode, it will continue to correctly interpret the data unless there is an error on the track.

We can now discuss the particular portions of a track in detail. The three gaps will be covered first. Unlike some other disk formats, the size of the three gap types will vary from drive to drive and even from track to track. During the initialization process, DOS will start with large gaps and keep making them smaller until an entire track can be written without overlapping itself. DOS makes sure that each gap type (see Figure 3.7) contains a minimum of four self-sync bytes. The result is fairly uniform gap sizes within each particular track.

Gap 1 is the first data written to a track during initialization. Its purpose is twofold. The gap originally consists of 128 bytes of self-sync, a large enough area to insure that all portions of a track will contain data. Since the speed of a particular drive may vary, the total length of the track in bytes is uncertain, and the percentage occupied by data is unknown. The initialization process is set up, however, so that even on drives of differing speeds, the last data field written will overlap Gap 1, providing continuity over the entire physical track. Care is taken to make sure the remaining portion of Gap 1 is at least as long as a typical Gap 3 (in practice its length is usually more than 400 sync bytes),

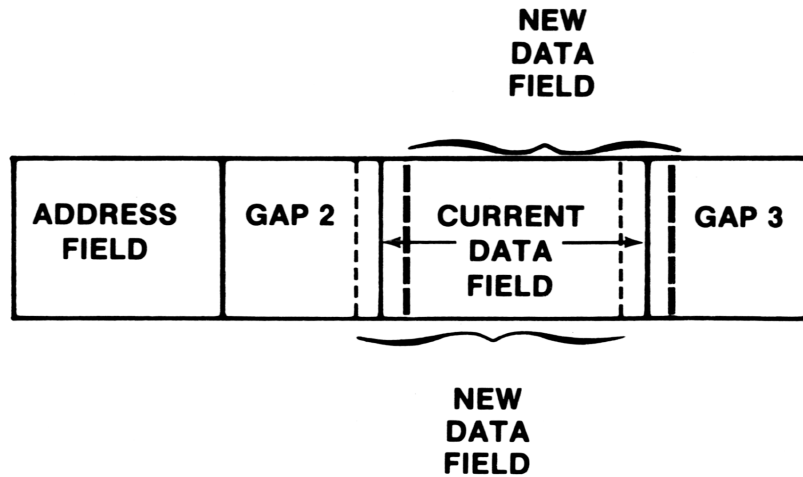


Figure 3.11 Gaps on Diskette

enabling it to serve as a Gap 3 type for Address Field number 0 (See Figure 3.7 for clarity).

Gap 2 appears after each Address Field and before each Data Field. Its length varies from five to ten bytes on a normal drive. The primary purpose of Gap 2 is to provide time for the information in an Address Field to be decoded by the computer before a read or write takes place. If the gap were too short, the beginning of the Data Field might spin past while DOS was still determining if this was the sector to be read. The 240 odd cycles that six self-sync bytes provide seems ample time to decode an address field. When a Data Field is written there is no guarantee that the write will occur in exactly the same spot each time. This is due to the fact that the drive which is rewriting the Data Field may not be the one which originally INITed or wrote it. Since the speed of the drives can vary, it is possible that the write could start in mid-byte. (See Figure 3.11) This is not a problem as long as the difference in positioning is not great. To insure the integrity of Gap 2, when writing a data field, five self-sync bytes are written prior to writing the Data Field itself. This serves two purposes. Since relatively little time is spent decoding an address field, the five bytes help place the Data Field near its original position. Secondly, and more importantly, the five self-sync bytes guarantee read-synchronization. It is probable that, in writing a Data Field, at least one sync byte will be destroyed. This is because, just as in reading bits on the track, the write may not begin on a byte boundary, thus altering an existing byte. Figure 3.12 illustrates this.

Gap 3 appears after each Data Field and before each Address Field. It is longer than Gap 2 and generally ranges from 14 to 24 bytes in length. It is quite similar in purpose to Gap 2. Gap 3 allows the additional time needed to

manipulate the data that has been read before the next sector is to be read. The length of Gap 3 is not as critical as that of Gap 2. If the following Address

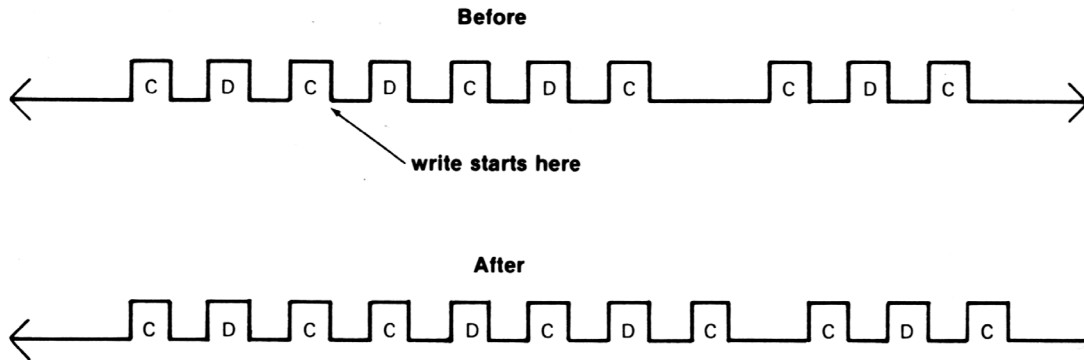
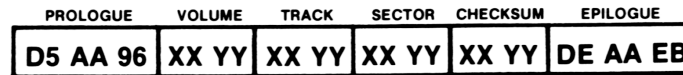


Figure 3.12 Writing Out of Sync

Field is missed, DOS can always wait for the next time it spins around under the read/write head, at most one revolution of the disk. Since Address Fields are never rewritten, there is no problem with this gap providing synchronization, since only the first part of the gap can be overwritten or damaged. (See Figure 3.11 for clarity)

An examination of the contents of the two types of fields is in order. The Address Field contains the “address” or identifying information about the Data Field which follows it. The volume, track, and sector number of any given sector can be thought of as its “address”, much like a country, city, and street number might identify a house. As shown previously in Figure 3.7, there are a number of components which make up the Address Field. A more detailed illustration is given in Figure 3.13.

The prologue consists of three bytes which form a unique sequence, found in no other component of the track. This fact enables DOS to locate an Address Field with almost no possibility of error. The three bytes are **\$D5**, **\$AA**, and **\$96**. The **\$D5** and **\$AA** are reserved (never written as data) thus insuring the uniqueness of the prologue. The **\$96**, following this unique string, indicates that the data following constitutes an Address Field (as opposed to a Data Field). The address information follows next, consisting of the volume, track, and sector number and a checksum. This information is absolutely essential for DOS to know where it is positioned on a particular diskette. The checksum is computed by exclusive-ORing the first three pieces of information, and is used to verify its integrity. Lastly follows the epilogue, which contains the three bytes **\$DE**, **\$AA** and **\$EB**. Oddly, the **\$EB** is always written during initialization but is never verified when an Address Field is read. The epilogue bytes are



ODD-EVEN ENCODED

DATA BYTE — $D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0$

XX — $1 D_7 1 D_5 1 D_3 1 D_1$

YY — $1 D_6 1 D_4 1 D_2 1 D_0$

Figure 3.13 Address Field

sometimes referred to as “bit-slip marks”, which provide added assurance that the drive is still in sync with the bytes on the disk. These bytes are probably unnecessary, but do provide a means of double checking.

The other field type is the Data Field. Much like the Address Field, it consists of a prologue, data, checksum, and an epilogue. (Refer to Figure 3.14) The prologue is different only in the third byte. The bytes are \$D5, \$AA, and \$AD, which again form a unique sequence, enabling DOS to locate the beginning of the sector data. The data consists of 342 bytes of encoded data. The encoding scheme used will be discussed in the next section. The data is followed by a checksum byte, used to verify the integrity of the data just read. The epilogue portion of the Data Field is absolutely identical to the epilogue in the Address Field and it serves the same function.

DATA FIELD ENCODING

Due to Apple’s hardware, it is not possible to read all 256 possible byte values from a diskette. This is not a great problem, but it does require that the data written to the disk be encoded. Three different techniques have been used. The

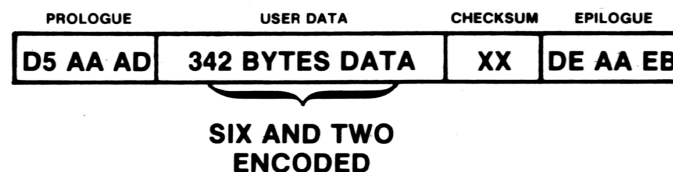


Figure 3.14 Data Field

first one, which is currently used in Address Fields, involves writing a data byte as two disk bytes, one containing the odd bits, and the other containing the even

bits. It would thus require 512 “disk” bytes for each 256 byte sector of data. Had this technique been used for sector data, no more than 10 sectors would have fit on a track. This amounts to about 88K of data per diskette, or roughly 72K of space available to the user; typical for 5 ¼-inch single density drives.

Fortunately, a second technique for writing data to diskette was devised that allows 13 sectors per track. This new method involved a “5 and 3” split of the data bits, versus the “4 and 4” mentioned earlier. Each byte written to the disk contains five valid bits rather than four. This requires 410 “disk” bytes to store a 256 byte sector. This latter density allows the now well-known 13 sectors per track format used by DOS 3 through DOS 3.2.1. The “5 and 3” scheme represented a hefty 33% increase over comparable drives of the day.

Currently, of course, DOS 3.3 features 16 sectors per track and provides

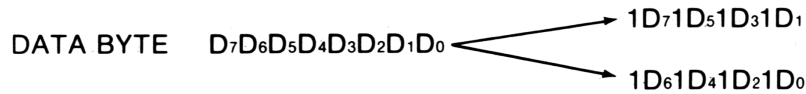


Figure 3.15 “4 and 4” Encoding

a 23% increase in disk storage over the 13 sector format. This was made possible by a hardware modification (the P6 PROM on the disk controller card) which allowed a “6 and 2” split of the data. The change was to the logic of the “state machine” in the P6 PROM, now allowing two consecutive zero bits in data bytes.

These three different encoding techniques will now be covered in some detail. The hardware for DOS 3.2.1 (and earlier versions of DOS) imposed a number of restrictions upon how data could be stored and retrieved. It required

$$\begin{array}{rcl}
 & D_7 & 1 \quad D_5 & 1 \quad D_3 & 1 \quad D_1 & 1 & & \text{(shifted left)} \\
 \text{AND} & 1 & D_6 & 1 \quad D_4 & 1 \quad D_2 & 1 \quad D_0 & & \\
 \hline
 & D_7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0
 \end{array}$$

Figure 3.16 “4 and 4” Decoding

that a disk byte have the high bit set and, in addition, no two consecutive bits could be zero. The odd-even “4 and 4” technique meets these requirements. Each data byte is represented as two bytes, one containing the even data bits and the other the odd data bits. Figure 3.15 illustrates this transformation. It should be noted that the unused bits are all set to one to guarantee meeting the two requirements.

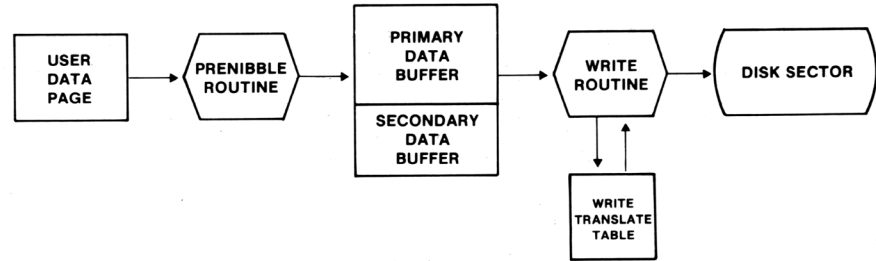


Figure 3.17 Memory Bits to Disk Bits Process

No matter what value the original data byte has, this technique insures that the high bit is set and that there can not be two consecutive zero bits. The “4 and 4” technique is used to store the information (volume, track, sector, checksum) contained in the Address Field. It is quite easy to decode the data, since the byte with the odd bits is simply shifted left and logically ANDed with the byte containing the even bits. This is illustrated in Figure 3.16.

It is important that the least significant bit contain a 1 when the odd-bits byte is left shifted. The entire operation is carried out in the **RDADR** subroutine at **\$B944** in DOS (48K).

The major difficulty with the above technique is that it takes up a lot of room on the track. To overcome this deficiency the “5 and 3” encoding technique was developed. It is so named because, instead of splitting the bytes in half, as in the odd-even technique, they are split five and three. A byte would have the form **000XXXXX**, where **X** is a valid data bit. The above byte could range in value from **\$00** to **\$1F**, a total of 32 different values. It so happens that there are 34 valid “disk” bytes, ranging from **\$AA** up to **\$FF**, which meet the two requirements (high bit set, no consecutive zero bits). Two bytes, **\$D5** and **\$AA**, were chosen as reserved bytes, thus leaving an exact mapping between five bit data bytes and eight bit “disk” bytes. The process of converting eight bit data bytes to eight bit “disk” bytes, then, is twofold. An overview is diagrammed in Figure 3.17.

First, the 256 bytes that will make up a sector must be translated to five bit bytes. This is done by the “prenibble” routine at **\$B800**. It is a fairly involved process, involving a good deal of bit rearrangement. Figure 3.18 shows the before and after of prenibbilizing. On the left is a buffer of eight bit data bytes, as passed to the **RWTS** subroutine package by DOS. Each byte in this buffer is represented by a letter (A, B, C, etc.) and each bit by a number (7 through 0). On the right side are the results of the transformation. The primary buffer contains five distinct areas of five bit bytes (the top three bits of the eight bit bytes zero-filled) and the secondary buffer contains three areas, graphically

illustrating the name “5 and 3”.

A total of 410 bytes are needed to store the original 256. This can be calculated by finding the total bits of data ($256 \times 8 = 2048$) and dividing that by the number of bits per byte ($2048/5 = 409.6$). (two bits are not used) Once this process is completed, the data is further transformed to make it valid “disk” bytes, meeting the disk’s requirements. This is much easier, involving a one to one look-up in the table given in Figure 3.19.

The Data Field has a checksum much like the one in the Address Field, used to verify the integrity

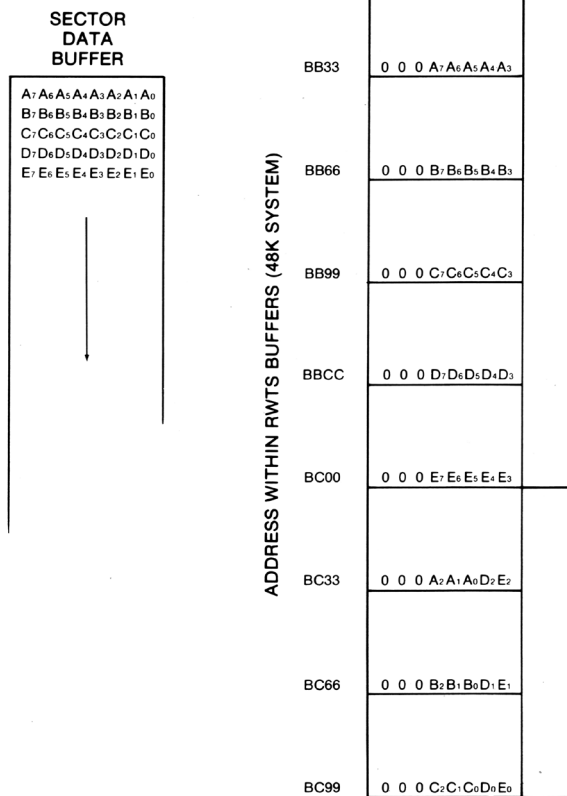


Figure 3.18 “5 and 3” Prenibblizing

of the data. It also involves exclusive-ORing the information, but, due to time constraints during reading bytes, it is implemented differently. The data is exclusive-ORED in pairs before being transformed by the look-up table in Figure 3.19. This can best be illustrated by Figure 3.20.

The reason for this transformation can be better understood by examining how the information is retrieved from the disk. The read routine must read a byte, transform it, and store it—all in under 32 cycles (the time taken to write a byte) or the information will be lost. By using the checksum computation to decode data, the transformation shown in Figure 3.20 greatly facilitates the time constraint. As the data is being read from a sector the accumulator contains the cumulative result of all previous bytes, exclusive-ORED together. The value of the accumulator after any exclusive-OR operation is the actual data byte for that point in the series. This process is diagrammed in Figure 3.21.*

* Figures 3.20 and 3.21 present the nibblizing process used by the “6 and 2” encoding technique.

However, the concept is the same for the “5 and 3” technique.

00 = AB	08 = BB	10 = DD	18 = F5
01 = AD	09 = BD	11 = DE	19 = F6
02 = AE	0A = BE	12 = DF	1A = F7
03 = AF	0B = BF	13 = EA	1B = FA
04 = B5	0C = D6	14 = EB	1C = FB
05 = B6	0D = D7	15 = ED	1D = FD
06 = B7	0E = DA	16 = EE	1E = FE
07 = BA	0F = DB	17 = EF	1F = FF

AA }
D5 } Reserved Bytes

Figure 3.19 "5 and 3" Write Translate Table

The third encoding technique, currently used by DOS 3.3, is similar to the "5 and 3". It was made possible by a change in the hardware which eased the requirements for valid data somewhat. The high bit must still be set, but now the byte may contain one (and only one) pair of consecutive zero bits. This allows a greater number of valid bytes and permits the use of a "6 and 2" encoding technique. A six bit byte would have the form 00XXXXXX and has values from \$00 to \$3F for a total of 64 different values. With the new, relaxed requirements for valid "disk" bytes there are 70 different bytes ranging in value from \$95 up to \$FF. After removing the two reserved bytes, \$AA and \$D5, there are still 68 "disk" bytes with only 64 needed. An additional requirement was introduced to force the mapping to be one to one, namely, that there must be at least two adjacent bits set, excluding bit 7. This eliminates four more bytes and produces exactly 64 valid "disk" values. The initial transformation is done by the prebible routine (still located at \$B800) and its results are shown in Figure 3.22.

A total of 342 bytes are needed, shown by finding the total number of bits (256 x 8 = 2048) and dividing by the number of bits per byte (2048 / 6 = 341.33). The transformation from the six bit bytes to valid data bytes is again performed by a one to one mapping shown in Figure 3.23. Once again, the stream of data bytes written to the diskette are a product of exclusive-ORs, exactly as with the "5 and 3" technique discussed earlier.

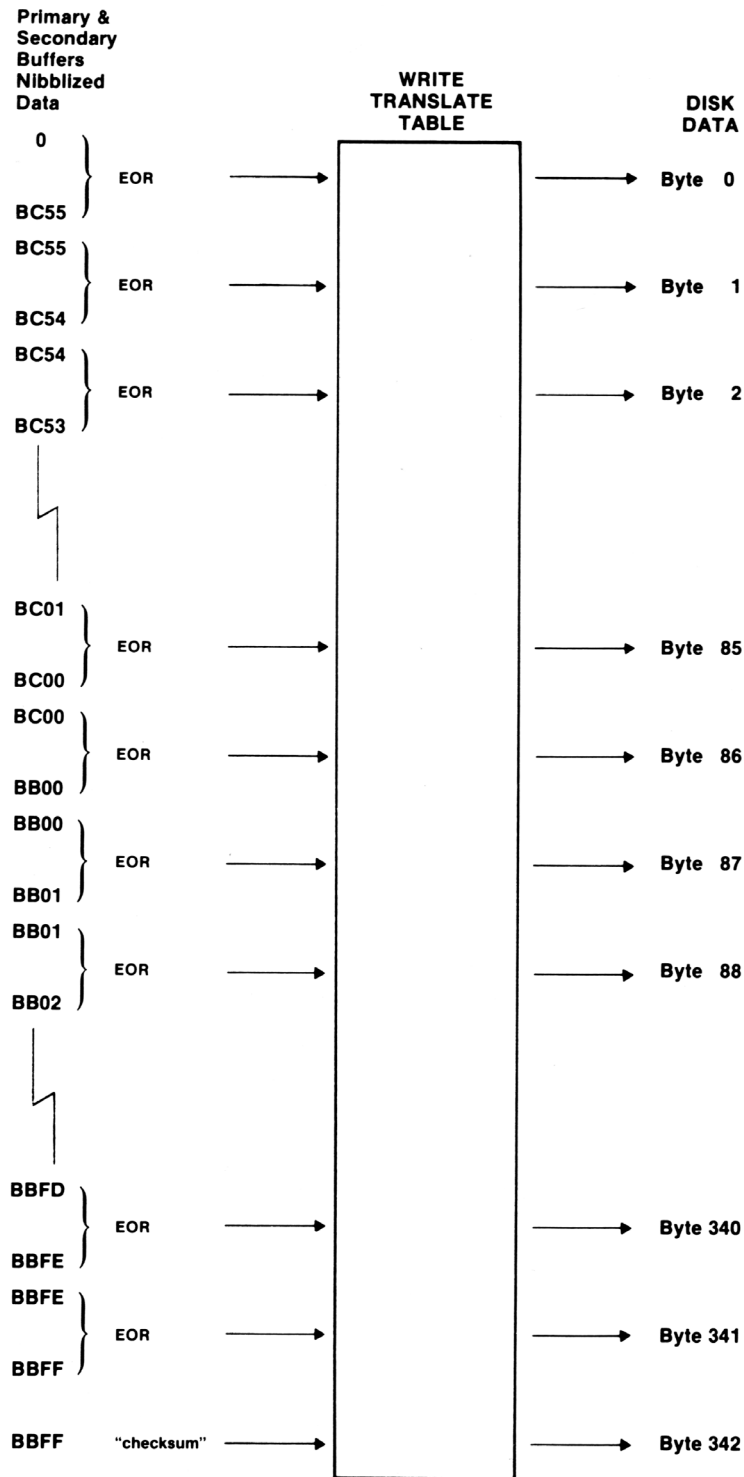


Figure 3.20 Writing to Diskette, DOS 3.3

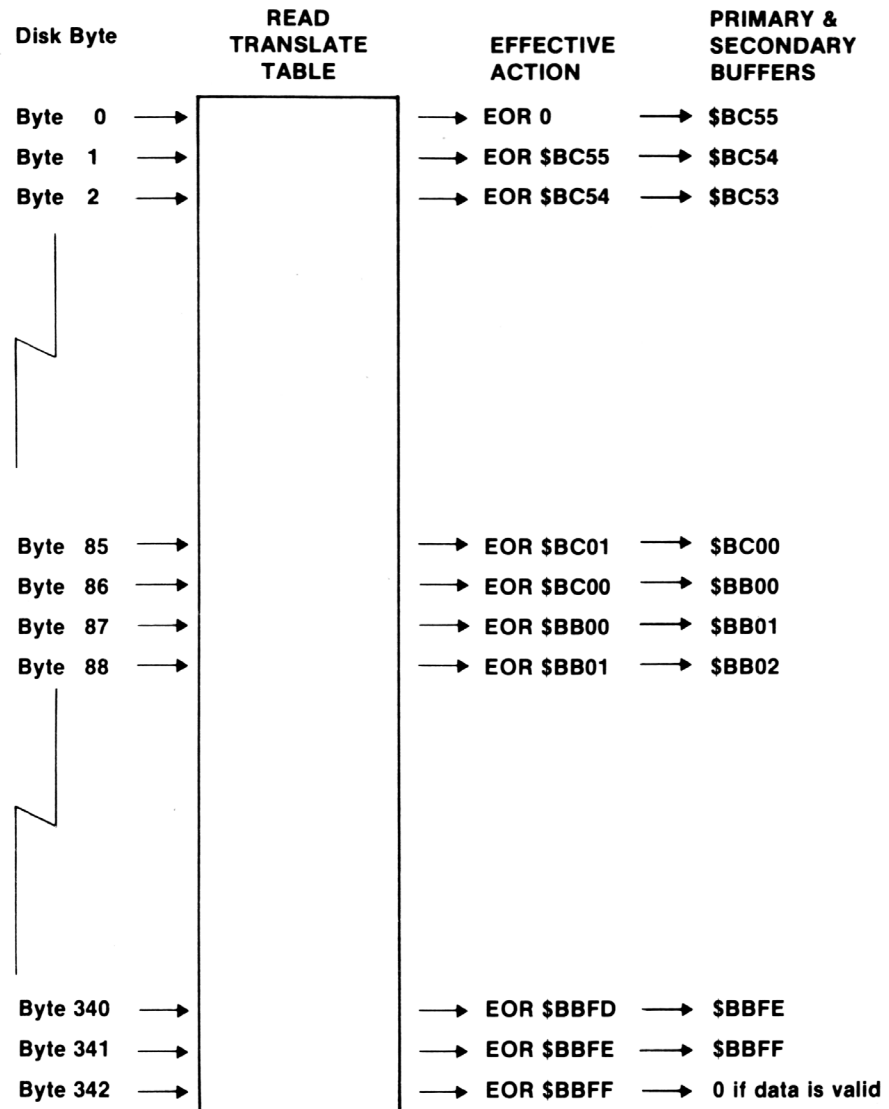


Figure 3.21 Reading From Diskette, DOS 3.3

SECTOR INTERLEAVING

Sector interleaving, or skewing, is the staggering of sectors on a track to maximize access speed. There is usually a delay between the time DOS reads or writes a sector and the time it is ready to read or write another. This delay depends upon the application program using the disk and can vary greatly. If sectors were stored on the track in sequential order, it would usually be necessary to wait a full revolution of the diskette before the next sector could

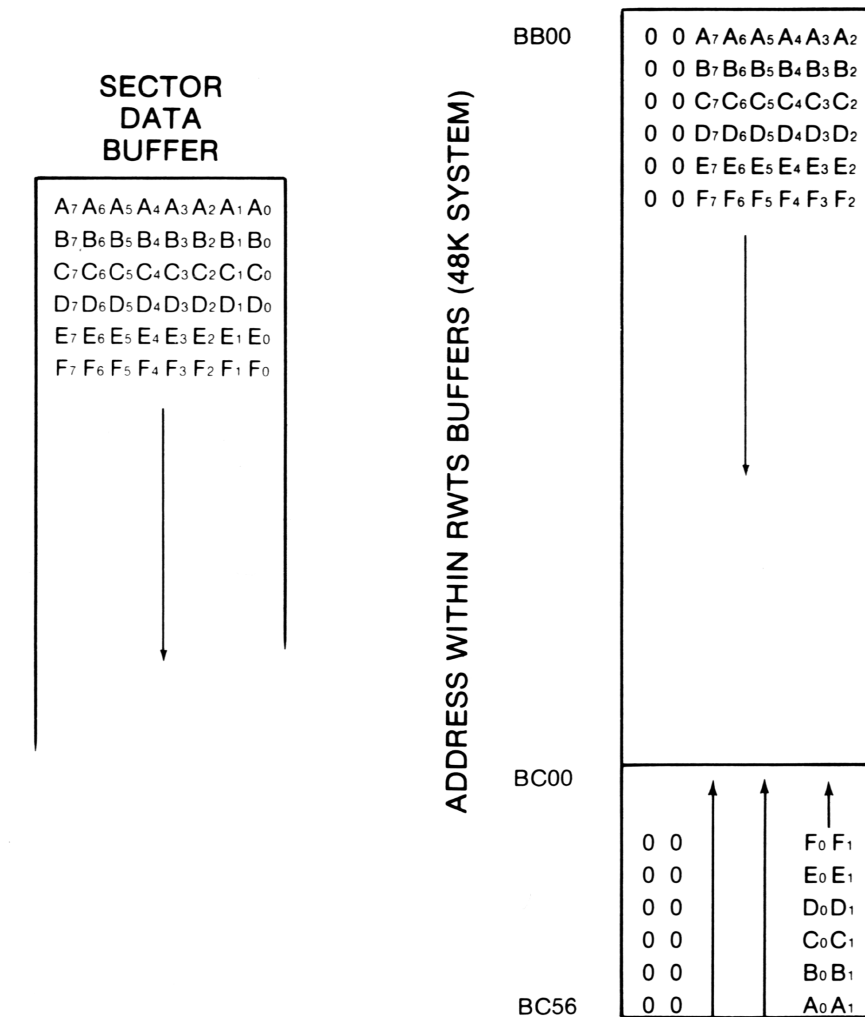


Figure 3.22 "6 and 2" Prebiting

be accessed. Ordering the sectors non-sequentially (skewing them) can provide improved access speeds.

On DOS 3.2.1 and earlier versions, the 13 sectors are physically skewed on the disk. During the boot operation, sectors are loaded from the diskette in ascending sequential order. However, files generally are stored in descending sequential order. As a result, no single interleaving scheme works well for both booting and sequentially accessing a file.

A different approach has been used in DOS 3.3 in an attempt to maximize performance. The skewing is now done in software. The 16 physical sectors are stored in ascending order (0, 1, 2, ..., 15) and are not physically skewed at all. A look-up table is used to translate a logical or soft sector number used by

00 = 96	10 = B4	20 = D6	30 = ED
01 = 97	11 = B5	21 = D7	31 = EE
02 = 9A	12 = B6	22 = D9	32 = EF
03 = 9B	13 = B7	23 = DA	33 = F2
04 = 9D	14 = B9	24 = DB	34 = F3
05 = 9E	15 = BA	25 = DC	35 = F4
06 = 9F	16 = BB	26 = DD	36 = F5
07 = A6	17 = BC	27 = DE	37 = F6
08 = A7	18 = BD	28 = DF	38 = F7
09 = AB	19 = BE	29 = E5	39 = F9
0A = AC	1A = BF	2A = E6	3A = FA
0B = AD	1B = CB	2B = E7	3B = FB
0C = AE	1C = CD	2C = E9	3C = FC
0D = AF	1D = CE	2D = EA	3D = FD
0E = B2	1E = CF	2E = EB	3E = FE
0F = B3	1F = D3	2F = EC	3F = FF

AA }
D5 } Reserved Bytes

Figure 3.23 “6 and 2” Write Translate Table

RWTS into the physical sector number found on the diskette. For example, if the logical sector number were a 2, this would be translated into the physical sector number 11 (\$0B). Thus RWTS treats physical sector 11 (\$0B) as sector 2 for all intents and purposes. This presents no problem if RWTS is used for disk access, but would become a consideration if access were made without RWTS. DOS 3.3 uses what we refer to as a “2 descending” skew.

In an attempt to eliminate the access differences between booting and reading files, another change was made to DOS 3.3. During the boot process, DOS is loaded backwards in descending sequential order into memory, just as files are accessed. However, due to differences in the delays for booting and reading files, no single skewing scheme is optimal.

It is interesting to point out that Pascal, Fortran, and CP/M diskettes all use software interleaving also. However, each uses a different sector order. Pascal and Fortran use a 2 ascending skew and CP/M diskettes use a 3 ascending skew. A comparison of these differences is presented in Figure 3.24.

Physical Sector	DOS 3.3 Sector	Pascal Sector	CP/M Sector
Ø	Ø	Ø	Ø
1	D	2	3
2	B	4	6
3	9	6	9
4	7	8	C
5	5	A	F
6	3	C	2
7	1	E	5
8	E	1	8
9	C	3	B
A	A	5	E
B	8	7	1
C	6	9	4
D	4	B	7
E	2	D	A
F	F	F	D

Figure 3.24 Comparison of Sector Interleaving

Chapter 4

Diskette Organization

As was described in Chapter 3, a 16 sector diskette consists of 560 data areas of 256 bytes each, called sectors. These sectors are arranged on the diskette in 35 concentric rings or tracks of 16 sectors each. The way DOS allocates these tracks of sectors is the subject of this chapter.

A file (be it Applesoft, Integer, Binary, or Text type) consists of one or more sectors containing data. Since the sector is the smallest unit of allocatable space on a diskette, a file will use up at least one sector even if it is less than 256 bytes long; the remainder of the sector is wasted. Thus, a file containing 400 characters (or bytes) of data will occupy one entire sector and 144 bytes of another with 112 bytes wasted. Knowing these facts, one would expect to be able to use up to 16 times 35 times 256 or 143,360 bytes of space on a diskette for files. Actually, the largest file that can be stored is about 126,000 bytes long.

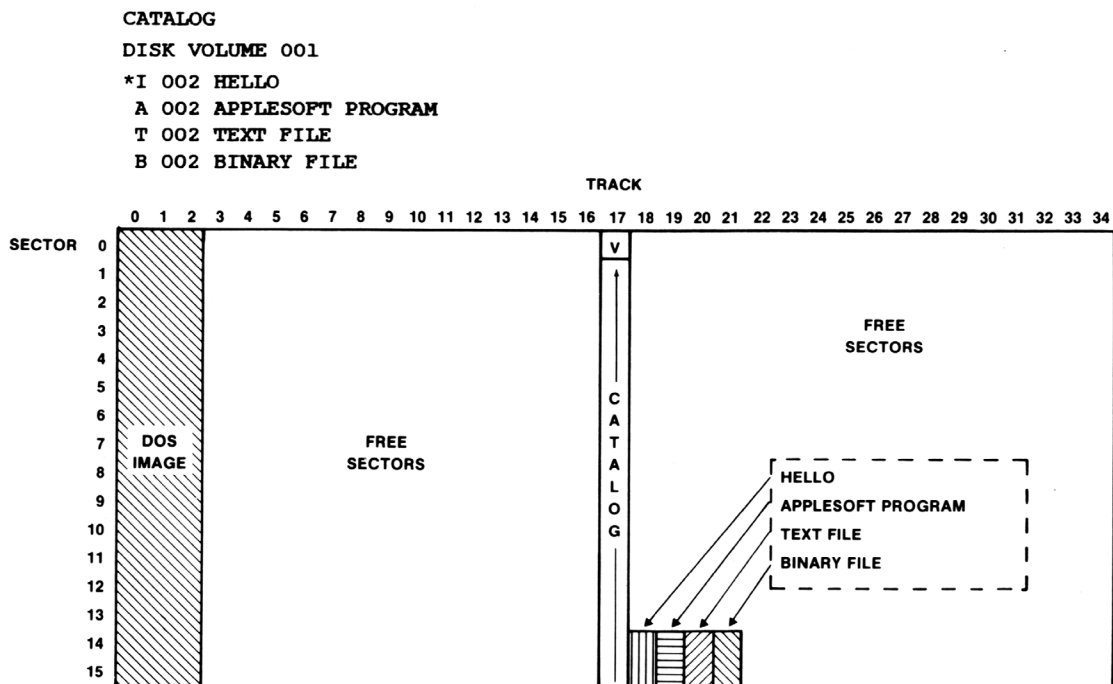


Figure 4.1 A Typical 16 Sector Diskette Map

The reason for this is that some of the sectors on the diskette must be used for what is called “overhead”.

Overhead sectors contain the image of DOS which is loaded when booting the diskette, a list of the names and locations of the files on the diskette, and an accounting of the sectors which are free for use with new files or expansions of existing files. An example of the way DOS uses sectors is given in Figure 4.1.

DISKETTE SPACE ALLOCATION

The map in Figure 4.1 shows that the first three tracks of each diskette are always reserved for the bootstrap image of DOS. In the exact center track (track 17) is the VTOC and catalog. The reason for placing the catalog here is simple. Since the greatest delay when using the disk is waiting for the arm to move from track to track, it is advantageous to minimize this arm movement whenever possible. By placing the catalog in the exact center track of the disk, the arm need never travel more than 17 tracks to get to the catalog track. As files are allocated on a diskette, they occupy the tracks just above the catalog track first. When the last track, track 34, has been used, track 16, the track adjacent and below the catalog, is used next, then 15, 14, 13, and so on, moving away from the catalog again, toward the DOS image tracks. If there are very few files on the diskette, they will all be clustered, hopefully, near the catalog and arm movement will be minimized. Additional space for a file, if it is needed, is first allocated in the same track occupied by the file. When that track is full, another track is allocated elsewhere on the disk in the manner described above.

HOW DOS IS STORED ON THE DISK

The following table shows how DOS is stored on the disk and which track and sector corresponds to which memory location:

Master Track/Sector	Slave Track/Sector	Master/Slave Track/Sector	Load Address	Relocated Address
DOS 3.2.1		DOS 3.3		
00, 00	00, 00	00, 00	3600	B600
00, 01	00, 01	00, 01	3700	B700
00, 02	00, 02	00, 02	3800	B800
00, 03	00, 03	00, 03	3900	B900
00, 04	00, 04	00, 04	3A00	BA00
00, 05	00, 05	00, 05	3B00	BB00
00, 06	00, 06	00, 06	3C00	BC00
00, 07	00, 07	00, 07	3D00	BD00

Master Track/Sector	Slave Track/Sector	Master/Slave Track/Sector	Load Address	Relocated Address
DOS 3.2.1		DOS 3.3		
00, 08	00, 08	00, 08	3E00	BE00
00, 09	00, 09	00, 09	3F00	BF00
00, 0A	—	00, 0A	1B00	—
00, 0B	—	00, 0B	1C00	—
00, 0C	00, 0A	00, 0C	1D00	9D00
01, 00	00, 0B	00, 0D	1E00	9E00
01, 01	00, 0C	00, 0E	1F00	9F00
01, 02	01, 00	00, 0F	2000	A000
01, 03	01, 01	01, 00	2100	A100
01, 04	01, 02	01, 01	2200	A200
01, 05	01, 03	01, 02	2300	A300
01, 06	01, 04	01, 03	2400	A400
01, 07	01, 05	01, 04	2500	A500
01, 08	01, 06	01, 05	2600	A600
01, 09	01, 07	01, 06	2700	A700
01, 0A	01, 08	01, 07	2800	A800
01, 0B	01, 09	01, 08	2900	A900
01, 0C	01, 0A	01, 09	2A00	AA00
02, 00	01, 0B	01, 0A	2B00	AB00
02, 01	01, 0C	01, 0B	2C00	AC00
02, 02	02, 00	01, 0C	2D00	AD00
02, 03	02, 01	01, 0D	2E00	AE00
02, 04	02, 02	01, 0E	2F00	AF00
02, 05	02, 03	01, 0F	3000	B000
02, 06	02, 04	02, 00	3100	B100
02, 07	02, 05	02, 01	3200	B200
02, 08	02, 06	02, 02	3300	B300
02, 09	02, 07	02, 03	3400	B400
02, 0A	02, 08	02, 04	3500	B500

THE VTOC

The Volume Table Of Contents is the “anchor” of the entire diskette. On any diskette accessible by any version of DOS, the VTOC sector is always in the same place; track 17, sector 0. (Some protected disks have the VTOC at another location and provide a special DOS which can find it.) Since files can end up anywhere on the diskette, it is through the VTOC anchor that DOS is able to find them. The VTOC of a diskette has the following format (all byte offsets are given in base 16, hexadecimal):

VOLUME TABLE OF CONTENTS (VTOC) FORMAT

BYTE	DESCRIPTION
00	Not used
01	Track number of first catalog sector
02	Sector number of first catalog sector
03	Release number of DOS used to INIT this diskette
04-05	Not used
06	Diskette volume number (1-254)
07-26	Not used
27	Maximum number of track/sector pairs which will fit in one file track/sector list sector (122 for 256 byte sectors)
28-2F	Not used
30	Last track where sectors were allocated
31	Direction of track allocation (+1 or -1)
32-33	Not used
34	Number of tracks per diskette (normally 35)
35	Number of sectors per track (13 or 16)
36-37	Number of bytes per sector (LO/HI format)
38-3B	Bit map of free sectors in track 0
3C-3F	Bit map of free sectors in track 1
40-43	Bit map of free sectors in track 2
	...
BC-BF	Bit map of free sectors in track 33
C0-C3	Bit map of free sectors in track 34
C4-FF	Bit maps for additional tracks if there are more than 35 tracks per diskette

BIT MAPS OF FREE SECTORS ON A GIVEN TRACK

A four byte binary string of ones and zeros, representing free and allocated sectors respectively. Hexadecimal sector numbers are assigned to bit positions as follows:

BYTE	SECTORS
+0	FEDC BA98
+1	7654 3210
+2 (not used)
+3 (not used)

Thus, if only sectors E and 8 are free and all others are allocated, the bit map will be:

41000000

If all sectors are free:

FFFF0000

An example of a VTOC sector is given in Figure 4.2. This VTOC corresponds to the map of the diskette given in Figure 4.1.

		A	B		C												
00	04	11	0F	03	00	00	01	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
						D											
20	00	00	00	00	00	00	7A	00	00	00	00	00	00	00	00	00	00
	E				F	G		H					I				
30	15	01	00	00	23	10	00	01	00	00	00	00	00	00	00	00	00
40	00	00	00	00	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	00
50	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	00
						J											
60	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	00
70	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	00	00	00	00	00
	K																
80	3F	FF	00	00	3F	FF	00	00	3F	FF	00	00	FF	FF	00	00	00
90	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	00
A0	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	00
B0	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	00
	L																
C0	FF	FF	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

- A** First CATALOG sector is on Track \$11 (17), sector \$0F (15)
- B** This is a DOS 3.3 disk
- C** Volume #1
- D** \$7A (122) Track/Sector pairs in a Track/Sector list
- E** Last allocated Track was \$15 (21), next will be \$16 (22)
- F** \$23 (35) tracks per disk, \$10 (16) sectors per track
- G** \$0100 (256) bytes per sector
- H** Track 0 has no available sectors
- I** Track 1 has no available sectors
- J** Track 12 has all sectors available
- K** Only sectors \$0E (14) and \$0F (15) are used on track \$12 (18)
- L** Track \$22 (34) has all sectors available

Figure 4.2 Example VTOC

THE CATALOG

In order for DOS to find a given file, it must first read the VTOC to find out where the first catalog sector is located. Typically, the catalog sectors for a diskette are the remaining sectors on track 17, following the VTOC sector. Of course, as long as a track/sector pointer exists in the VTOC and the VTOC is located at track 17, sector 0, DOS does not really care where the catalog resides. Figure 4.3 diagrams the catalog track. The figure shows the track/sector pointer in the VTOC at bytes 01 and 02 as an arrow pointing to track 17 (11 in hexadecimal) sector F. The last sector in the track is the first catalog sector and describes the first seven files on the diskette. Each catalog sector has a track/sector pointer in the same position (bytes 01 and 02) which points to the next catalog sector. The last catalog sector (sector 1) has a zero pointer to indicate that there are no more catalog sectors in the chain.

In each catalog sector up to seven files may be listed and described. Thus, on a typical DOS 3.3 diskette, the catalog can hold up to 15 times 7, or 105 files. A catalog sector is formatted as described on the following page.

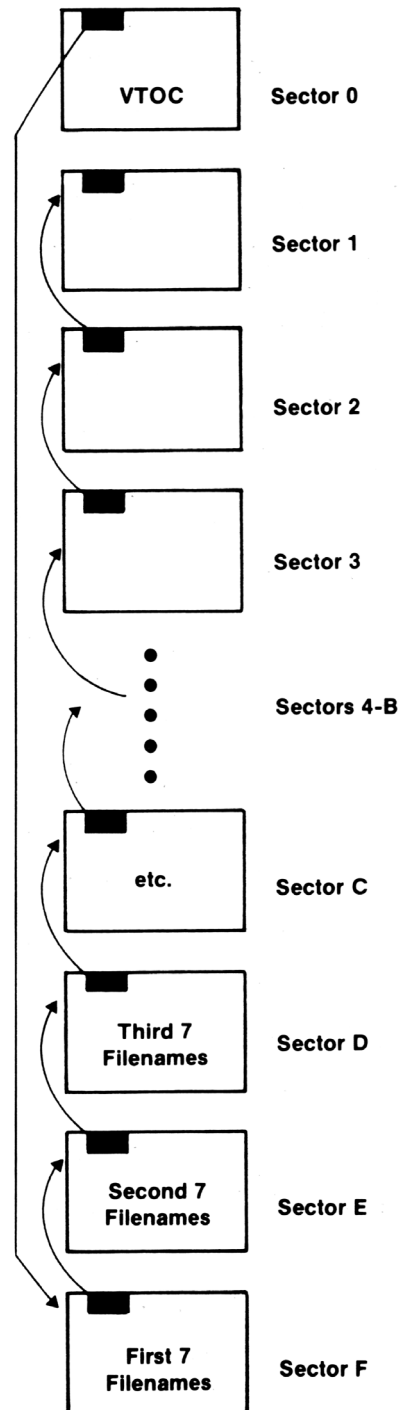


Figure 4.3 Track 17, the CATALOG Track

CATALOG SECTOR FORMAT

BYTE	DESCRIPTION
00	Not used
01	Track number of next catalog sector (usually 11 hex)
02	Sector number of next catalog sector
03-0A	Not used
0B-2D	First file descriptive entry
2E-50	Second file descriptive entry
51-73	Third file descriptive entry
74-96	Fourth file descriptive entry
97-B9	Fifth file descriptive entry
BA-DC	Sixth file descriptive entry
DD-FF	Seventh file descriptive entry

FILE DESCRIPTIVE ENTRY FORMAT

RELATIVE

BYTE	DESCRIPTION
00	Track of first track/sector list sector. If this is a deleted file, this byte contains a hex FF and the original track number is copied to the last byte of the file name field (Byte 20). If this byte contains a hex 00, the entry is assumed to never have been used and is available for use. (This means track 0 can never be used for data even if the DOS image is "wiped" from the diskette.)
01	Sector of first track/sector list sector
02	File type and flags: Hex 80+file type - file is locked 00+file type - file is not locked 00 - Text file 01 - Integer BASIC file 02 - Applesoft BASIC file 04 - Binary file 08 - S type file 10 - Relocatable object module file 20 - A type file 40 - B type file (thus, 84 is a locked Binary file, and 90 is a locked R type file)
03-20	File name (30 characters)
21-22	Length of file in sectors (LO/HI format). The CATALOG command will only format the LO byte of this length giving 1-255 but a full 65,535 may be stored here.

	00	00		.	Unused
	01	11 0E		.	Next CATALOG sector is Track \$11 (17), Sector \$0E (14)
	03	00 00 00 00 00 00 00 00		Unused
File Entry 1	0B	12 0F		.	First Track/Sector list for this file is Track \$12 (18), Sector \$0F (15)
	0D	81		.	\$81 = Locked, Integer BASIC program file
	0E	C8 C5 CC CC CF A0 A0 A0	H E L L O	.	File name is "HELLO"
	16	A0 A0 A0 A0 A0 A0 A0 A0		.	
	1E	A0 A0 A0 A0 A0 A0 A0 A0		.	
	26	A0 A0 A0 A0 A0 A0		.	
	2C	02 00		.	File Length is \$0002 (2) sectors
					First Track/Sector list for this file is Track \$13 (19), Sector \$0F (15)
File Entry 2	2E	13 0F		.	\$02 = Unlocked Applesoft program file
	30	02		.	File name is "APPLESOFT PROGRAM"
	31	C1 D0 D0 CC C5 D3 CF C6	A P P L E S O F	.	
	39	D4 A0 D0 D2 CF C7 D2 C1	T P R O G R A	.	
	41	CD A0 A0 A0 A0 A0 A0 A0	M	.	
	49	A0 A0 A0 A0 A0 A0		.	
	4F	02 00		.	File Length is \$0002 (2) sectors
					First Track/Sector list for this file is Track \$14 (20), Sector \$0F (15)
File Entry 3	51	14 0F		.	\$00 = Unlocked Text file
	53	00		.	File name is "TEXT FILE"
	54	D4 C5 D8 D4 A0 C6 C9 CC	T E X T F I L	.	
	5C	C5 A0 A0 A0 A0 A0 A0 A0	E	.	
	64	A0 A0 A0 A0 A0 A0 A0 A0		.	
	6C	A0 A0 A0 A0 A0 A0		.	
	72	02 00		.	File Length is \$0002 (2) sectors
					First Track/Sector list for this file is Track \$15 (21), Sector \$0F (15)
File Entry 4	74	15 0F		.	\$04 = Unlocked Binary image file
	76	04		.	File name is "BINARY FILE"
	77	C2 C9 CE C1 D2 D9 A0 C6	B I N A R Y F	.	
	7F	C9 CC C5 A0 A0 A0 A0 A0	I L E	.	
	87	A0 A0 A0 A0 A0 A0 A0 A0		.	
	8F	A0 A0 A0 A0 A0 A0		.	
	95	02 00		.	File Length is \$0002 (2) sectors
	97	00		.	
	98	00 00 00 00 00 00 00 00		.	Rest of CATALOG sector is empty
	A0	00 00 00 00 00 00 00 00		.	
	A8	00 00 00 00 00 00 00 00		.	
	B0	00 00 00 00 00 00 00 00		.	
	B8	00 00 00 00 00 00 00 00		.	
	C0	00 00 00 00 00 00 00 00		.	Alternating Shaded/Unshaded areas show where file entries would be placed if more files are added to the diskette.
	C8	00 00 00 00 00 00 00 00		.	
	D0	00 00 00 00 00 00 00 00		.	
	D8	00 00 00 00 00 00 00 00		.	
	E0	00 00 00 00 00 00 00 00		.	
	E8	00 00 00 00 00 00 00 00		.	
	F0	00 00 00 00 00 00 00 00		.	
	F8	00 00 00 00 00 00 00 00		.	

Figure 4.4 Typical CATALOG Sector

Figure 4.4 is an example of a typical catalog sector. In this example there are only four files on the entire diskette, so only one catalog sector was needed to describe them. There are four entries in use and three entries which have never been used and contain zeros.

THE TRACK/SECTOR LIST

Each file has associated with it a "Track/Sector List" sector. This sector contains a list of track/sector pointer pairs which sequentially list the data sectors which make up the file. The file descriptive entry in the catalog sector points to this T/S List sector which, in turn, points to each sector in the file. This concept is diagramed in Figure 4.5.

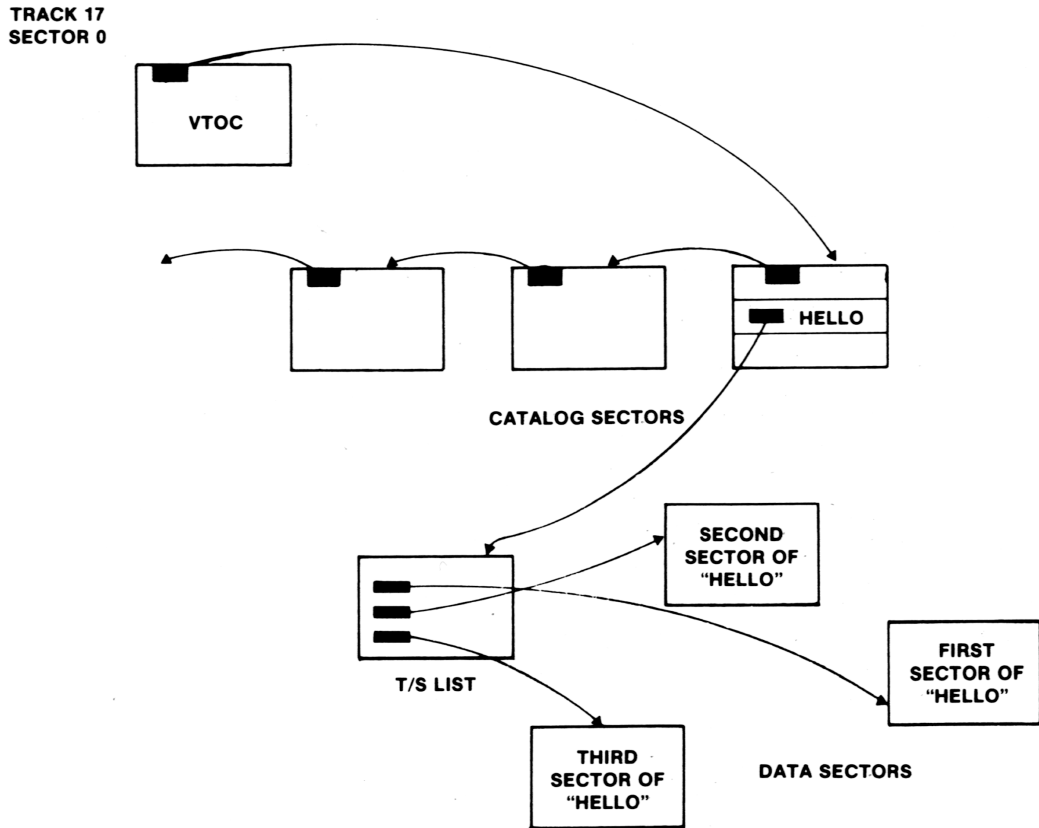


Figure 4.5 Path DOS Must Follow to Find a File

The format of a Track/Sector List sector is given below. Note that since even a minimal file requires one T/S List sector and one data sector, the least number of sectors a non-empty file can have is 2. Also, note that a very large file, having more than 122 data sectors, will need more than one Track/Sector List to hold all the Track/Sector pointer pairs.

TRACK/SECTOR LIST FORMAT

BYTE	DESCRIPTION
00	Not used
01	Track number of next T/S List sector if one was needed or zero if no more T/S List sectors.
02	Sector number of next T/S List sector (if present).
03-04	Not used
05-06	Sector offset in file of the first sector described by this list.
07-0B	Not used

0C-0D Track and sector of first data sector or zeros
0E-0F Track and sector of second data sector or zeros
10-FF Up to 120 more Track/Sector pairs

A sequential file will end when the first zero T/S List entry is encountered. A random file, however, can have spaces within it which were never allocated and therefore have no data sectors allocated in the T/S List. This distinction is not always handled correctly by DOS. The **VERIFY** command, for instance, stops when it gets to the first zero T/S List entry and can not be used to verify some random organization text files.

```

00      00      Not used
01      00      Track number of next T/S list (0 = no more T/S lists)
02      00      Sector number of next T/S list (0 = no more T/S lists)
03      00 00   Not used
05      00 00   Sector offset of file in this T/S list (0 = beginning of file)
07      00 00 00 00 00 Not used
0C      13 0E   First Track and Sector of this file
0E      13 0D   Second Track and Sector of this file
10      13 0C   Third Track and Sector of this file
12      13 0B   Fourth Track and Sector of this file
14      13 0A   Final Track and Sector of this file
           Rest of list is zeroes (no more T/Ss)
16      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F0      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

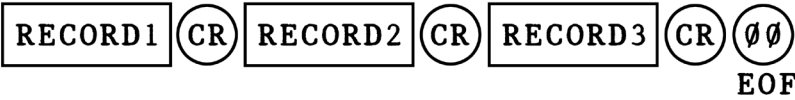
Figure 4.6 Example Track/Sector List

An example T/S List sector is given in Figure 4.6. The example file (**HELLO**, from our previous examples) has only one data sector, since it is less than 256 bytes in length. Counting this data sector and the T/S List sector, **HELLO** is 2 sectors long, and this will be the value shown when a **CATALOG** command is done.

Following the Track/Sector pointer in the T/S List sector, we come to the first data sector of the file. As we examine the data sectors, the differences between the file types become apparent. All files (except, perhaps, a random Text file) are considered to be continuous streams of data, even though they must be broken up into 256 byte chunks to fit in sectors on the diskette. Although these sectors are not necessarily contiguous (or next to each other on the diskette), by using the Track/Sector List, DOS can read each sector of

the file in the correct order so that the programmer need never know that the data was broken up into sectors at all.

Sequential Text File Format



00	D2	C5	C3	CF	D2	C4	B1	8D	R	E	C	O	R	D	1	.
08	D2	C5	C3	CF	D2	C4	B2	8D	R	E	C	O	R	D	2	.
10	D2	C5	C3	CF	D2	C4	B3	8D	R	E	C	O	R	D	3	.
18	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00
28	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00
38	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00
48	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00
58	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00
68	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00
78	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00
88	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00
98	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00
A8	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00
B8	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00
C8	00	00	00	00	00	00	00	00
D0	00	00	00	00	00	00	00	00
D8	00	00	00	00	00	00	00	00
E0	00	00	00	00	00	00	00	00
E8	00	00	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00
F8	00	00	00	00	00	00	00	00

Figure 4.7 Example Sequential Text File Sector

TEXT FILES

The Text data type is the least complicated file data structure. It consists of one or more records, separated from each other by carriage return characters (hex 8Ds). This structure is diagrammed and an example file is given in Figure 4.7. Usually, the end of a Text file is signaled by the presence of a hex 00 or the lack of any more data sectors in the T/S List for the file. As mentioned earlier, if the file has random organization, there may be hex 00s imbedded in the data and even missing data sectors in areas where nothing was ever written. In this case, the only way to find the end of the file is to scan the Track/Sector List for the last non-zero Track/Sector pair. Since carriage return characters and hex 00s have special meaning in a Text type file, they can not be part of the data itself. For this reason, and to make the data accessible to BASIC, the data can only contain printable or ASCII characters (alphabetic, numerics

Binary File Format on Disk



00	03 08	Program start address (\$0803)
02	4E 12	File length (\$124E = 4,686 bytes)

Rest of Sector is a Binary Memory Image																
04	20	44	08	8D	94	13	A9	00	8D	AD	13	BA				
10	8E	1C	13	20	8B	08	20	DA	08	AD	AC	13	A2	16	20	2C
20	0B	D0	09	20	62	08	20	59	0D	4C	3E	08	20	F1	09	20
30	62	08	20	40	0B	AD	AB	13	D0	05	A2	0F	20	CD	0A	20
40	1D	0B	4C	0E	08	A9	80	85	76	85	D9	85	33	38	AD	D2
50	03	E9	07	E9	1E	30	01	60	20	58	FC	A2	15	20	CD	0A
60	4C	D3	03	A2	01	BD	1F	13	0A	0A	0A	0A	9D	21	13	CA
70	10	F3	20	DC	03	84	00	85	01	20	E3	03	84	02	85	03
80	20	2D	10	20	E8	0F	A9	00	8D	AB	13	60	A9	00	85	22
90	20	58	FC	A2	16	20	CD	0A	A2	17	20	CD	0A	20	8E	FD
A0	A0	00	A2	1F	20	CD	0A	B9	AF	13	20	ED	FD	A2	20	20
B0	CD	0A	98	18	69	28	AA	20	CD	0A	C8	C0	09	D0	E3	20
C0	8E	FD	A2	21	20	CD	0A	20	6F	FD	AD	00	02	A2	00	20
D0	2C	0B	D0	B8	8D	AC	13	8C	AE	13	60	20	58	FC	A9	0F
E0	85	24	AD	AE	13	18	69	28	AA	20	CD	0A	A9	03	85	22
F0	20	58	FC	AD	AC	13	A2	0A	20	2C	0B	D0	07	A9	00	8D

Figure 4.8 Example Binary File Sector

or special characters, see p. 8 in the *Apple II Reference Manual*). This restriction makes processing of a Text file slower and less efficient in the use of disk space than with a Binary type file, since each digit must occupy a full byte in the file.

BINARY FILES

The structure of a Binary type file is shown in Figure 4.8. An exact copy of the memory involved is written to the disk sector(s), preceded by the memory address where it was found and the length (a total of four bytes). The address and length (in low order, high order format) are those given in the **A** and **L** keywords from the **BSAVE** command which created the file. Notice that DOS writes one extra byte to the file. This does not matter too much since **BLOAD** and **BRUN** will only read the number of bytes given in the length field. (Of course, if you **BSAVE** a multiple of 256 bytes, a sector will be wasted because of this error) DOS could be made to **BLOAD** or **BRUN** the binary image at a different address either by providing the **A** (address) keyword when the command is entered, or by changing the address in the first two bytes of the file on the diskette.

APPLESOFT AND INTEGER FILES

A BASIC program, be it Applesoft or Integer, is saved to the diskette in a way that is similar to **BSAVE**. The format of an Applesoft file type is given in Figure 4.9 and that of Integer BASIC in 4.10. When the **SAVE** command is typed, DOS determines the location of the BASIC program image in memory and its length. Since a BASIC program is always loaded at a location known to the BASIC interpreter, it is not necessary to store the address in the file as with a Binary file. The length is stored, however, as the first two bytes, and is followed by the image from memory. Notice that, again, DOS incorrectly writes an additional byte, even though it will be ignored by **LOAD**. The memory image of the program consists of program lines in an internal format which is made up of what are called “tokens”. A treatment of the structure of a BASIC program as it appears in memory is outside the scope of this manual, but a breakdown of the example Integer BASIC program is given in Figure 4.10.

OTHER FILE TYPES (S, R, NEW A, NEW B)

Additional file types have been defined within DOS as can be seen in the file descriptive entry format, shown earlier. No DOS commands at present use

Applesoft File Format on Disk

L / H

Program Memory Image...

```
1Ø PRINT "[CTRL-D] OPEN TEXT FILE"
2Ø PRINT "[CTRL-D] WRITE TEXT FILE"
3Ø PRINT "1,2,3,4"
4Ø PRINT "[CTRL-D] CLOSE TEXT FILE"
5Ø END
```

Ø Ø

5F	Ø Ø
----	-----

Program is \$005F (95) bytes long

Applesoft Program

[illegible]

Figure 4.9 Example Applesoft File Sector

Integer BASIC File Format on Disk

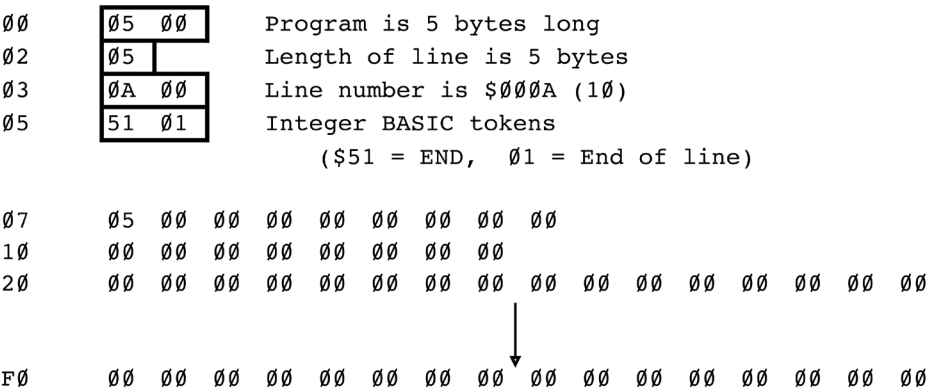
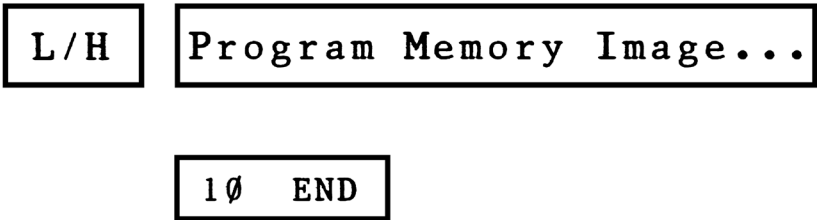


Figure 4.10 Example Integer BASIC File Sector

these additional types so their eventual meaning is anybody's guess. The R file type, however, has been used with the DOS Toolkit assembler for its output file, a relocatable object module. This file type is used with a special form of Binary file which can contain the memory image of a machine language program which may be relocated anywhere in the machine based on additional information stored with the image itself. The format for this type of file is given in the documentation accompanying the DOS Toolkit. It is recommended that if the reader requires more information about R files he should refer to that documentation.

EMERGENCY REPAIRS

From time to time the information on a diskette can become damaged or lost. This can create various symptoms, ranging from mild side effects, such as the disk not booting, to major problems, such as an input/output (I/O) error in the catalog. A good understanding of the format of a diskette, as described previously, and a few program tools can allow any reasonably sharp Apple II user to patch up most errors on diskettes.

A first question would be, “how do errors occur?”. The most common cause of an error is a worn or physically damaged diskette. Usually, a diskette will warn you that it is wearing out by producing “soft errors”. Soft errors are I/O errors which occur only randomly. You may get an I/O error message when you catalog a disk one time and have it catalog correctly if you try again. When this happens, the smart programmer immediately copies the files on the aged diskette to a brand new one and discards the old one or keeps it as a backup.

Another cause of damaged diskettes is the practice of hitting the RESET key to abort the execution of a program which is accessing the diskette. Damage will usually occur when the RESET signal comes just as data is being written onto the disk. Powering the machine off just as data is being written to the disk is also a sure way to clobber a diskette. Of course, real hardware problems in the disk drive or controller card and ribbon cable can cause damage as well.

If the damaged diskette can be cataloged, recovery is much easier. A damaged DOS image in the first three tracks can usually be corrected by running the **MASTER CREATE** program against the diskette or by copying all the files to another diskette. If only one file produces an I/O error when it is **VERIFIED**, it may be possible to copy most of the sectors of the file to another diskette by skipping over the bad sector with an assembler program which calls **RWTS** in DOS or with a BASIC program (if the file is a Text file). Indeed, if the problem is a bad checksum (see Chapter 3) it may be possible to read the bad sector and ignore the error and get most of the data.



When you forget to backup your disks

An I/O error usually means that one of two conditions has occurred. Either a bad checksum was detected on the data in a sector, meaning that one or more bytes is bad; or the sectoring is clobbered such that the sector no longer even exists on the diskette. If the latter is the case, the diskette (or at the very least, the track) must be reformatted, resulting in a massive loss of data. Although DOS can be patched to format a single track, it is usually easier to copy all readable sectors from the damaged diskette to another formatted diskette and then reconstruct the lost data there.



When you know you made a backup

Disk utilities, such as Quality Software's *Bag of Tricks*, allow the user to read and display the contents of sectors. *Bag of Tricks* will also allow you to modify the sector data and rewrite it to the same or another diskette. If you do not have *Bag of Tricks* or another commercial disk utility, you can use the **ZAP** program in Appendix A of this book. The **ZAP** program will read any track/sector on an unprotected diskette into memory, allowing the user to examine it or modify the data and then, optionally, rewrite it to a diskette. Using such a program is very important when learning about diskette formats and when fixing clobbered data.

Using **ZAP**, a bad sector within a file can be localized by reading each track/sector listed in the T/S List sector for the file. If the bad sector is a catalog sector, the pointers of up to seven files may be lost. When this occurs, a search of the diskette can be made to find T/S List sectors which do not correspond to any files listed in the remaining "good" catalog sectors. As these sectors are found, new file descriptive entries can be made in the damaged sector which point to these T/S Lists. When the entire catalog is lost, this process can take hours, even with a good understanding of the format of DOS diskettes. Such an endeavor should only be undertaken if there is no other way to recover the data. Of course the best policy is to create backup copies of important files periodically to simplify recovery. More information on the above procedures is given in Appendix A.

A less significant form of diskette clobber, but very annoying, is the loss of free sectors. Since DOS allocates an entire track of sectors at a time while a file is open, hitting RESET can cause these sectors to be marked in use in

the VTOC even though they have not yet been added to any T/S List. These lost sectors can never be recovered by normal means, even when the file is deleted, since they are not in its T/S List. The result is a **DISK FULL** message before the diskette is actually full. To reclaim the lost sectors it is necessary to compare every sector listed in every T/S List against the VTOC bit map to see if there are any discrepancies. There are utility programs which will do this automatically but the best way to solve this problem is to copy all the files on the diskette to another diskette (note that **FID** must be used, not **COPY**, since **COPY** copies an image of the diskette, bad VTOC and all).

If a file is deleted it can usually be recovered, providing that additional sector allocations have not occurred since it was deleted. If another file was created after the **DELETE** command, DOS might have reused some or all of the sectors of the old file. The catalog can be quickly **ZAPped** to move the track number of the T/S List from byte 20 of the file descriptive entry to byte 0. The file should then be copied to another disk and then the original deleted so that the VTOC freespace bit map will be updated.



Freshly initialized disks are the best

Chapter 5

The Structure of Dos

DOS MEMORY USE

DOS is an assembly language program which is loaded into RAM memory when the user boots his disk. If the diskette booted is a master diskette, the DOS image is loaded into the last possible part of RAM memory, dependent upon the size of the actual machine on which it is run. By doing this, DOS fools the active BASIC into believing that there is actually less RAM memory on the machine than there is. On a 48K Apple II with DOS active, for instance, BASIC believes that there is only about 38K of RAM. DOS does this by adjusting HIMEM after it is loaded to prevent BASIC from using the memory DOS is occupying. If a slave diskette is booted, DOS is loaded into whatever RAM it occupied when the slave diskette was **INITIALIZED**. If the slave was created on a 16K Apple, DOS will be loaded in the 6 to 16K range of RAM, even if the machine now has 48K. In this case, the Apple will appear, for all intents and purposes, to have only 6K of RAM. If the slave was created on a 48K system, it

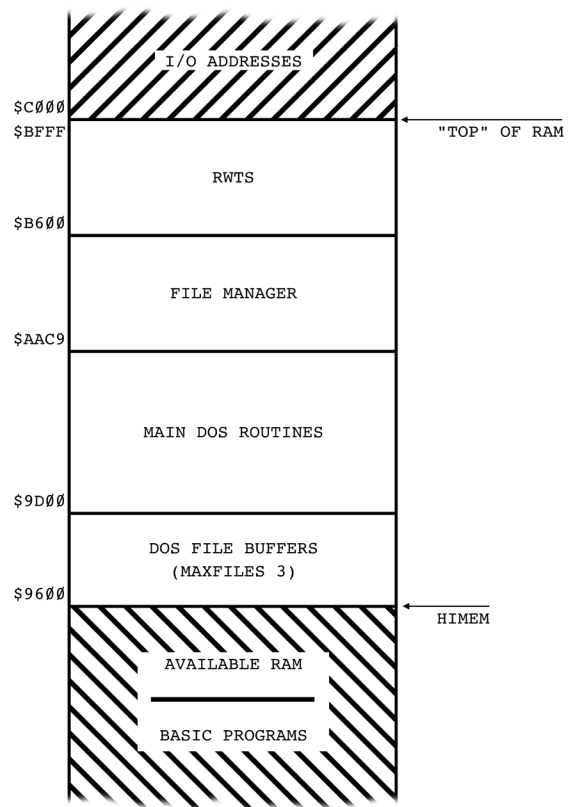


Figure 5.1 DOS Memory Use (48K Apple)

will not boot on less than 48K since the RAM DOS occupied does not exist on a smaller machine.

A diagram of DOS's memory for a 48K Apple II is given in Figure 5.1. As can be seen, there are four major divisions to the memory occupied by DOS. The first 1.75K is used for file buffers. With the default of **MAXFILES** 3, there are three file buffers set aside here. Each buffer occupies 595 bytes and corresponds to one potentially open file. File buffers are also used by DOS to **LOAD** and **SAVE** files, etc. If **MAXFILES** is changed from 3, the space occupied by the file buffers also changes. This affects the placement of HIMEM, moving it up or down with fewer or more buffers respectively.

The 3.5K above the file buffers is occupied by the main DOS routines. It is here that DOS's executable machine language code begins. The main routines are responsible for initializing DOS, interfacing to BASIC, interpreting commands, and managing the file buffers. All disk functions are passed on via subroutine calls to the file manager.

The file manager, occupying about 2.8K, is a collection of subroutines which perform almost any function needed to access a disk file. Functions include: **OPEN**, **CLOSE**, **READ**, **WRITE**, **POSITION**, **DELETE**, **CATALOG**, **LOCK**, **UNLOCK**, **RENAME**, **INIT**, and **VERIFY**. Although the file manager is a subroutine of DOS it may also be called by a user written assembly language program which is not part of DOS. This interface is generalized through a group of vectors in page 3 of RAM and is documented in the next chapter.

The last 2.5K of DOS is the Read/Write Track/Sector (RWTS) package. RWTS is the next step lower in protocol from the file manager - in fact it is called as a subroutine by the file manager. Where the file manager deals with files, RWTS deals with tracks and sectors on the diskette. A typical call to RWTS would be to read track 17 sector 0 or to write 256 bytes of data in memory onto track 5 sector E. An external interface is also provided for access to RWTS from a user written assembly language program and is described in the next chapter.

THE DOS VECTORS IN PAGE 3

In addition to the approximately 10K of RAM occupied by DOS in high memory, DOS maintains a group of what are called "vectors" in page 3 of low memory (\$300 through \$3FF). These vectors allow access to certain places within the DOS collection of routines via a fixed location (\$3D0 for instance). Because DOS may be loaded in various locations, depending upon the size of the machine and whether a slave or master diskette is booted, the addresses

of the externally callable subroutines within DOS will change. By putting the addresses of these routines in a vector at a fixed location, dependencies on DOS's location in memory are eliminated. The page 3 vector table is also useful in locating subroutines within DOS which may not be in the same memory location for different versions of DOS. Locations \$300 through \$3CF were used by earlier versions of DOS during the boot process to load the Boot 1 program but are used by DOS 3.3 as a data buffer and disk code translate table. Presumably, this change was made to provide more memory for the first bootstrap loader (more on this later). The vector table itself starts at \$3D0.

DOS VECTOR TABLE (\$3D0-\$3FF)

ADDR	USAGE
3D0	A JMP (jump or GOTO) instruction to the DOS warmstart routine. This routine reenters DOS but does not discard the current BASIC program and does not reset MAXFILES or other DOS environmental variables.
3D3	A JMP to the DOS coldstart routine. This routine reinitializes DOS as if it was rebooted, clearing the current BASIC file and resetting HIMEM.
3D6	A JMP to the DOS file manager subroutine to allow a user written assembly language program to call it.
3D9	A JMP to the DOS Read/Write Track/Sector (RWTS) routine to allow user written assembly language programs to call it.
3DC	A short subroutine which locates the input parameter list for the file manager to allow a user written program to set up input parameters before calling the file manager.
3E3	A short subroutine which locates the input parameter list for RWTS to allow a user written program to set up input parameters before calling RWTS.
3EA	A JMP to the DOS subroutine which "reconnects" the DOS intercepts to the keyboard and screen data streams.
3EF	A JMP to the routine which will handle a BRK machine language instruction. This vector is only supported by the Autostart ROM. Normally the vector contains the address of the monitor ROM subroutine which displays the registers.
3F2	LO/HI address of routine which will handle RESET for the Autostart ROM. Normally the DOS restart address is stored here but the user may change it if he wishes to handle RESET himself.
3F4	Power-up byte. Contains a "funny complement" of the RESET address with a \$A5. This scheme is used to determine if the machine was just powered up or if RESET was pressed. If a power-up occurred, the Autostart ROM ignores the address at 3F2 (since it has never been initialized) and attempts to boot a

diskette. To prevent this from happening when you change \$3F2 to handle your own RESETs, EOR (exclusive OR) the new value at \$3F2 with a \$A5 and store the result in the power-up byte.

- 3F5 A JMP to a machine language routine which is to be called when the 'E' feature is used in Applesoft.
- 3F8 A JMP to a machine language routine which is to be called when a control-Y is entered from the monitor.
- 3FB A JMP to a machine language routine which is to be called when a non-maskable interrupt occurs.
- 3FE LO/HI address of a routine which is to be called when a maskable interrupt occurs.

WHAT HAPPENS DURING BOOTING

When an Apple is powered on its memory is essentially devoid of any programs. In order to get DOS running, a diskette is “booted”. The term “boot” refers to the process of bootstrap loading DOS into RAM. Bootstrap loading involves a series of steps which load successively bigger pieces of a program until all of the program is in memory and is running. In the case of DOS, bootstrapping occurs in four stages. The location of these stages on the diskette and a memory map are given in Figure 5.2 and a description of the bootstrap process follows.

The first boot stage (let's call it Boot 0) is the execution of the ROM on the disk controller card. When the user types **PR#6** or **C600G** or **6(ctrl)P**, for instance, control is transferred to the disk controller ROM on the card in slot 6. This ROM is a machine language program of about 256 bytes in length. When executed, it “recalibrates” the disk arm by pulling it back to track 0 (the “clackety-clack” noise that is heard) and then reads sector 0 from track 0 into RAM memory at location \$800 (DOS 3.3. Earlier versions used \$300). Once this sector is read, the first stage boot jumps (GOTO's) \$800 which is the second stage boot (Boot 1).

Boot 1, also about 256 bytes long, uses part of the Boot 0 ROM as a subroutine and, in a loop, reads the next nine sectors on track 0 (sectors 1 through 9) into RAM. Taken together, these sectors contain the next stage of the bootstrap process, Boot 2. Boot 2 is loaded in one of two positions in memory, depending upon whether a slave or a master diskette is being booted. If the diskette is a slave diskette, Boot 2 will be loaded 9 pages (256 bytes per page) below the end of the DOS under which the slave was **INITed**. Thus, if the slave was created on a 32K DOS, Boot 2 will be loaded in the RAM from \$7700 to \$8000. If a master diskette is being booted, Boot 2 will be loaded in the same place as for a 16K slave (\$3700 to \$4000). In the process of

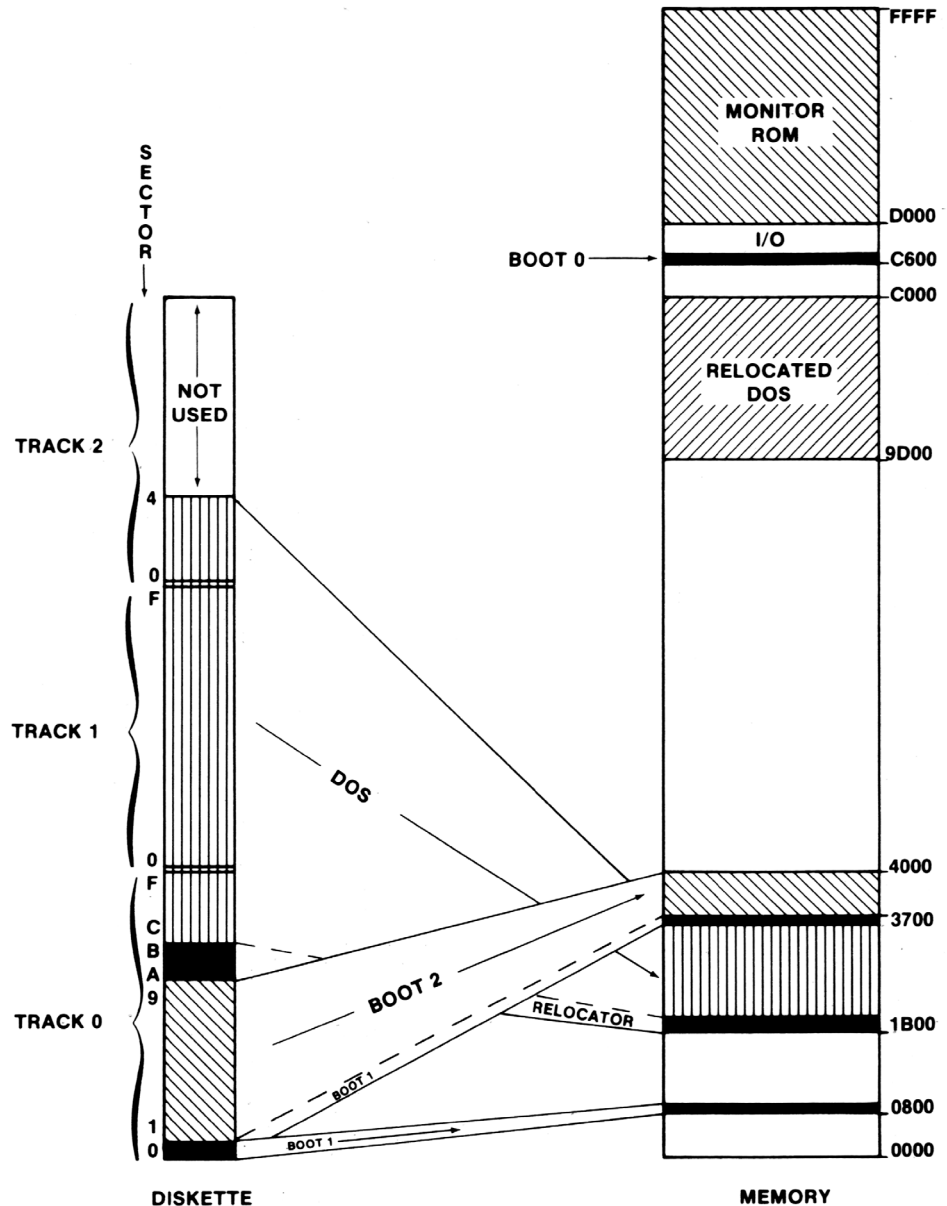


Figure 5.2 Bootstrap Process

loading Boot 2, Boot 1 is loaded a second time in the page in memory right below Boot 2 (\$3600 for a master diskette). This is so that, should a new diskette be INITed, a copy of Boot 1 will be available in memory to be written to its track 0 sector 0. When Boot 1 is finished loading Boot 2, it jumps there to begin execution of the next stage of the bootstrap.

Boot 2 consists of two parts: a loader “main program”; and the RWTS subroutine package. Up to this point there has been no need to move the disk arm since all of the necessary sectors have been on track 0. Now, however, more sectors must be loaded, requiring arm movement to access additional tracks. Since this complicates the disk access, RWTS is called by the Boot 2 loader to move the arm and read the sectors it needs to load the last part of the bootstrap, DOS itself. Boot 2 now locates track 2 sector 4 and reads its contents into RAM just below the image of Boot 1 (this would be at `$3500` for a master diskette). In a loop, Boot 2 reads 26 more sectors into memory, each one 256 bytes before the last. The last sector (track 0 sector A) is read into `$1B00` for a master diskette. The 27 sectors which were read are the image of the DOS main routines and the file manager. With the loading of these routines, all of DOS has been loaded into memory. At this point, the bootstrap process for a slave diskette is complete and a jump is taken to the DOS coldstart address. If the diskette is a master, the image of DOS is only valid if the machine is a 16K Apple II. If more memory is present, the DOS image must be relocated into the highest possible RAM present in the machine. To do this, the master version of Boot 2 jumps to a special relocation program at `$1B03`. This relocater is 512 bytes in length and was automatically loaded as the two lowest pages of the DOS image. (In the case of a slave diskette, these pages contain binary zeros.) The relocater determines the size of the machine by systematically storing and loading on high RAM memory pages until it finds the last valid page. It then moves the DOS image from `$1D00` to its final location (`$9D00` for 48K) and, using tables built into the program, it modifies the machine language code so that it will execute properly at its new home. The relocater then jumps to the high memory copy of DOS and the old image is forgotten.

The DOS boot is completed by the DOS coldstart routine. This code initializes DOS, making space for the file buffers, setting HIMEM, building the page 3 vector table, and running the `HELLO` program.

Previous versions of DOS were somewhat more complicated in the implementation of the bootstrap. In these versions, Boot 1 was loaded at `$300` and it, in turn, loaded Boot 2 at `$3600`, as does version 3.3. Unlike 3.3, however, 27 sectors of DOS were not always loaded. If the diskette was a slave diskette, only 25 sectors were loaded, and, on 13 sector diskettes, this meant the DOS image ended either with sector 8 or sector A of track 2 depending upon whether the diskette was a slave or master. In addition, Boot 1 had a different form of nibbilization (see chapter 3) than any other sector on the diskette, making its raw appearance in memory at `$3600` non-executable.

The various stages of the bootstrap process will be covered again in greater detail in Chapter 8, *DOS Program Logic*.



You gotta tell me: how did you
boot from the external //c drive?



Let me get this straight: you have
a Floppy Emu and a Booti card?

Chapter 6

Using DOS from Assembly Language

CAVEAT

This chapter is aimed at the advanced assembly language programmer who wishes to access the disk without resorting to the **PRINT** statement scheme used with BASIC. Accordingly, the topics covered here may be beyond the comprehension (at least for the present) of a programmer who has never used assembly language.

DIRECT USE OF DISK DRIVE

It is often desirable or necessary to access the Apple's disk drives directly from assembly language, without the use of DOS. This is done using a section of 16 addresses that are latched toggles, interfacing directly to the hardware. There are eight two byte toggles that essentially represent pulling a TTL line high or low. Applications which could use direct disk access range from a user written operating system to DOS-independent utility programs. The device address assignments are given in Figure 6.1.

ADDRESS	LABEL	DESCRIPTION

\$C080	PHASEOFF	Stepper motor phase 0 off.
\$C081	PHASEON	Stepper motor phase 0 on.
\$C082	PHASE1OFF	Stepper motor phase 1 off.
\$C083	PHASE1ON	Stepper motor phase 1 on.
\$C084	PHASE2OFF	Stepper motor phase 2 off.
\$C085	PHASE2ON	Stepper motor phase 2 on.
\$C086	PHASE3OFF	Stepper motor phase 3 off.
\$C087	PHASE3ON	Stepper motor phase 3 on.
\$C088	MOTOROFF	Turn motor off.
\$C089	MOTORON	Turn motor on.
\$C08A	DRV0EN	Engage drive 1.
\$C08B	DRV1EN	Engage drive 2.
\$C08C	Q6L	Strobe Data Latch for I/O.
\$C08D	Q6H	Load Data Latch.

\$C08E	Q7L	Prepare latch for input.
\$C08F	Q7H	Prepare latch for output.
	Q7L followed by Q6L	= Read
	Q7L followed by Q6H	= Sense Write Protect
	Q7H followed by Q6L	= Write
	Q7H followed by Q6H	= Load Write Latch

FIGURE 6.1 DOS HARDWARE ADDRESSES

The addresses are slot dependent and the offsets are computed by multiplying the slot number by 16. In hexadecimal this works out nicely and we can add the value **\$s0** (where *s* is the slot number) to the base address. If we wanted to engage disk drive number 1 in slot number 6, for example, we would add **\$60** to **\$C08A** (device address assignment for engaging drive 1) for a result of **\$C0EA**. However, since it is generally desirable to write code that is not slot dependent, one would normally use **\$C08A,X** (where the X register contains the value **\$s0**).

In general, the above addresses need only be accessed with any valid 6502 instruction. However, in the case of reading and writing bytes, care must be taken to insure that the data will be in an appropriate register. All of the following would engage drive number 1. (Assume slot number 6)

```
LDA $C0EA
BIT $C08A,X (where X-reg contains $60)
CMP $C08A,X (where X-reg contains $60)
```

Below are typical examples demonstrating the use of the device address assignments. For more examples, see Appendix A. Slot 6 is assumed and the X-register contains **\$60**.

STEPPER PHASE OFF/ON

Basically, each of the four phases (0-3) must be turned on and then off again. Done in ascending order, this moves the arm inward. In descending order, this moves the arm outward. The timing between accesses to these locations is critical, making this a non-trivial exercise. It is recommended that the **SEEK** command in RWT'S be used to move the arm. See the section on using RWT'S immediately following.

MOTOR OFF/ON

LDA \$C088,X	Turn motor off.
LDA \$C089,X	Turn motor on.

NOTE: A sufficient delay should be provided to allow the motor time to come up to speed. Shugart recommends one second, but DOS is able to reduce this delay by watching the read latch until data starts to change.

ENGAGE DRIVE 1/2

LDA \$C08A,X	Engage drive 1.
LDA \$C08B,X	Engage drive 2.

READ A BYTE

```

READ  LDA $C08C,X
      BPL READ

```

NOTE: \$C08E,X must already have been accessed to assure Read mode. The loop is necessary to ensure that the accumulator will contain valid data. If the data latch does not yet contain valid data the high bit will be zero.

SENSE WRITE PROTECT

LDA \$C08D,X	
LDA \$C08E,X	Sense write protect.
BMI ERROR	If high bit set, protected.

LOAD AND WRITE A BYTE

LDA DATA	
STA \$C08D,X	Write load.
ORA \$C08C,X	Write byte.

NOTE: \$C08F,X must already have been accessed to ensure Write mode and a 100 microsecond delay should be invoked before writing.

Due to hardware constraints, data bytes must be written in 32 cycle loops. Below is an example for an immediate load of the accumulator, followed by a write. Timing is so critical that different routines may be necessary, depending on how the data is to be accessed, and code can not cross memory page boundaries without an adjustment.

	LDA #\$D5	(2 cycles)
	JSR WRITE9	(6)
	LDA #\$AA	(2)
	JSR WRITE9	(6)
	.	
	.	
	.	
WRITE9	CLC	(2)
WRITE7	PHA	(3)
	PLA	(4)
WRITE	STA \$C08D,X	(5)
	ORA \$C08C,X	(4)
	RTS	(6)

CALLING READ/WRITE TRACK/SECTOR (RWTS)

Read/Write Track/Sector (RWTS) exists in every version of DOS as a collection of subroutines, occupying roughly the top third of the DOS program. The interface to RWTS is standardized and thoroughly documented by Apple and may be called by a program running outside of DOS.

There are two subroutines which must be called or whose function must be performed.

JSR \$3E3 When this subroutine is called, the Y and A registers are loaded with the address of the Input/Output control Block (IOB) used by DOS when accessing RWTS. The low order part of the address is in Y and the high order part in A. This subroutine should be called to locate the IOB and the results may be stored in two zero page locations to allow storing values in the IOB and retrieving output values after a call to RWTS. Of course, you may set up your own IOB as long as the Y and A registers point to your IOB upon calling RWTS.

JSR \$3D9 This is the main entry to the RWTS routine. Prior to making this call, the Y and A registers must be loaded with the address of an IOB describing the operation to be performed. This may be done by first calling \$3E3 as described above. The IOB must contain appropriate information as defined in the list on the facing page (offsets are given in hexadecimal):

INPUT/OUTPUT CONTROL BLOCK - GENERAL FORMAT

BYTE	DESCRIPTION
00	Table type, must be \$01
01	Slot number times 16 (s0: s=slot. Example: \$60)
02	Drive number (\$01 or \$02)
03	Volume number expected (\$00 matches any volume)
04	Track number (\$00 through \$22)
05	Sector number (\$00 through \$0F)
06-07	Address (LO/HI) of the Device Characteristics Table
08-09	Address (LO/HI) of the 256 byte buffer for READ/WRITE
0A	Not used
0B	Byte count for partial sector (\$00 for 256 bytes)
0C	Command code \$00 = SEEK \$01 = READ \$02 = WRITE \$04 = FORMAT
0D	Return code - The processor CARRY flag is set upon return from RWTS if there is a non-zero return code: \$00 = No errors \$08 = Error during initialization \$10 = Write protect error \$20 = Volume mismatch error \$40 = Drive error \$80 = Read error (obsolete)
0E	Volume number of last access (must be initialized)
0F	Slot number of last access * 16 (must be initialized)
10	Drive number of last access (must be initialized)
DEVICE CHARACTERISTICS TABLE	
BYTE	DESCRIPTION
00	Device type (should be \$00 for Disk II)
01	Phases per track (should be \$01 for Disk II)
02-03	Motor on time count (should be \$EFD8 for Disk II)

NOTE: RWTS uses zero-page location \$48, which is also used by the Apple monitor to hold the P-register value. Location \$48 should be set to zero after each call to RWTS.

RWTS IOB BY CALL TYPE

SEEK	Move disk arm to desired track
Input:	Byte 00 - Table type (\$01)
	01 - Slot number * 16 (s0: s=slot)
	02 - Drive number (\$01 or \$02)
	04 - Track number (\$00 through \$22)

	06/07	- Pointer to the DCT
	0C	- Command code for SEEK (\$00)
	0F	- Slot number of last access * 16
	10	- Drive number of last access
Output: Byte	0D	- Return code (See previous definition)
	0F	- Current Slot number * 16
	10	- Current Drive number

READ	Read a sector into a specified buffer	
Input: Byte	00	- Table type (\$01)
	01	- Slot number * 16 (s0: s=slot)
	02	- Drive number (\$01 or \$02)
	03	- Volume number (\$00 matches any volume)
	04	- Track number (\$00 through \$22)
	05	- Sector number (\$00 through \$0F)
	06/07	- Pointer to the DCT
	08/09	- Pointer to 256 byte user data buffer
	0B	- Byte count per sector (\$00)
	0C	- Command code for READ (\$01)
	0E	- Volume number of last access
	0F	- Slot number of last access * 16
	10	- Drive number of last access
Output: Byte	0D	- Return code (See previous definition)
	0E	- Current Volume number
	0F	- Current Slot number * 16
	10	- Current Drive number

WRITE	Write a sector from a specified buffer	
Input: Byte	00	- Table type (\$01)
	01	- Slot number * 16 (s0: s=slot)
	02	- Drive number (\$01 or \$02)
	03	- Volume number (\$00 matches any volume)
	04	- Track number (\$00 through \$22)
	05	- Sector number (\$00 through \$0F)
	06/07	- Pointer to the DCT
	08/09	- Pointer to 256 byte user data buffer
	0B	- Byte count per sector (\$00)
	0C	- Command code for WRITE (\$02)
	0E	- Volume number of last access
	0F	- Slot number of last access * 16
	10	- Drive number of last access
Output: Byte	0D	- Return code (See previous definition)
	0E	- Current Volume number
	0F	- Current Slot number * 16
	10	- Current Drive number

FORMAT	Initialize the diskette (does not put DOS on disk, create a VTOC/CATALOG, or store HELLO program)	
Input: Byte	00	- Table type (\$01)
	01	- Slot number * 16 (s0: s=slot)
	02	- Drive number (\$01 or \$02)
	03	- Volume number (\$00 will default to 254)

	06/07	- Pointer to the DCT
	0C	- Command code for FORMAT (\$04)
	0E	- Volume number of last access
	0F	- Slot number of last access * 16
	10	- Drive number of last access
Output: Byte	0D	- Return code (See previous definition)
	0E	- Current Volume number
	0F	- Current Slot number * 16
	10	- Current Drive number

CALLING THE DOS FILE MANAGER

The DOS file manager exists in every version of DOS as a collection of subroutines occupying approximately the central third of the DOS program. The interface to these routines is generalized in such a way that they may be called by a program running outside of DOS. The definition of this interface has never been published by Apple (or anyone else, for that matter) but since the calls can be made through fixed vectors, and, the format of the parameter lists passed have not changed in all the versions of DOS, these routines may be relied upon as “safe”. Indeed, the new **FID** utility program uses these routines to process files on the diskette.

There are two subroutines which must be called in order to access the file manager.

JSR \$3DC When this subroutine is called, the Y and A registers are loaded with the address of the file manager parameter list. The low order part of the address is in Y and the high order part in A. This subroutine must be called at least once to locate this parameter list and the results may be stored in two zero page locations to allow the programmer to set input values in the parameter list and to locate output values there after file manager calls.

JSR \$3D6 This is the main entry to the file manager. Prior to making this call the parameter list, located using the call described above, must be completed appropriately, depending upon the type of call, and the X register must be set to either zero or non-zero as follows:

X = 0	- If file is not found, allocate it
X # 0	- If file is not found, do not allocate one

Normally, X should be zero on an **OPEN** call for a new file and non-zero for all other call types.

Three buffers must be provided to the file manager by the programmer, allocated in memory. These buffers, together, occupy 557 bytes of RAM, and

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
OPEN	01		RECORD LENGTH OR 0000		V	D	S	FILE TYPE	FILE NAME ADDRESS		RETURN CODE (OUTPUT)							
CLOSE	02																	
READ	03	S U B C O D E	RECORD NUMBER	BYTE OFFSET	RANGE LENGTH	ONE DATA BYTE	RANGE DATA ADDR.		FILE NAME ADDRESS									
WRITE	04																	
DELETE	05			V	D	S			FILE NAME ADDRESS									
CATALOG	06				D	S												
LOCK	07			V	D	S			FILE NAME ADDRESS									
UNLOCK	08			V	D	S												
RENAME	09		NEW NAME ADDRESS	V	D	S												
POSITION	0A		RECORD NUMBER	BYTE OFFSET														
INIT	0B	DOS PAGE NO.		V	D	S												
VERIFY	0C			V	D	S			FILE NAME ADDRESS									

Figure 6.2 File Manager Parameter List Required Input

must be passed to the file manager each time their associated file is used. A separate set of these buffers must be maintained for each open file. DOS maintains buffers for this purpose, as described in earlier chapters, in high RAM. These buffers may be “borrowed” from DOS if care is taken to let DOS know about it. A method for doing this will be outlined later.

The general format of the file manager parameter list is as follows:

FILE MANAGER PARAMETER LIST - GENERAL FORMAT

BYTE	DESCRIPTION
00	Call type: 01=OPEN 05=DELETE 09=RENAME 02=CLOSE 06=CATALOG 0A=POSITION 03=READ 07=LOCK 0B=INIT 04=WRITE 08=UNLOCK 0C=VERIFY
01	Sub-call type for READ or WRITE: 00=No operation (ignore call entirely) 01=READ or WRITE one byte 02=READ or WRITE a range of bytes 03=POSITION then READ or WRITE one byte 04=POSITION then READ/WRITE a range
02-09	Parameters specific to the call type used. See FILE MANAGER PARAMETER LIST BY CALL TYPE below.
0A	Return code (note: not all return codes can occur for any call type). The processor CARRY flag is set upon return from the file manager if there is a non-zero return code: 00=No errors 01=Not used (“LANGUAGE NOT AVAILABLE”) 02=Bad call type 03=Bad sub-call type (greater than four) 04=WRITE PROTECTED 05=END OF DATA 06=FILE NOT FOUND (was allocated if X=0) 07=VOLUME MISMATCH 08=DISK I/O ERROR 09=DISK FULL 0A=FILE LOCKED
0B	Not used
0C-0D	Address of a 45 byte buffer which will be used by the file manager to save its status between calls. This area is called the file manager workarea and need not be initialized by the caller but the space must be provided and this two byte address field initialized. (addresses are in low/high order format)
0E-0F	Address of a 256 byte buffer which will be used by the file manager to maintain the current Track/Sector List sector for the open file. Buffer itself need not be

initialized by the caller.

10-11 Address of a 256 byte buffer which will be used by the file manager to maintain the data sector buffer.

Buffer need not be initialized by the caller.

FILE MANAGER PARAMETER LIST BY CALL TYPE

OPEN Locates or creates a file. A call to POSITION should follow every OPEN.

Input: Byte 00 - 01
 02/03 - Fixed record length or 0000 if variable
 04 - Volume number or 00 for any volume
 05 - Drive number to be used (01 or 02)
 06 - Slot number to be used (01-07)
 07 - File type (used only for new files)
 \$00 = TEXT
 \$01 = INTEGER BASIC
 \$02 = APPLESOFT BASIC
 \$04 = BINARY
 \$08 = RELOCATABLE
 \$10 = S TYPE FILE
 \$20 = A TYPE FILE
 \$40 = B TYPE FILE
 08/09 - Address of file name (30 characters)
 (Low/high format)
 0C/0D - Address of file manager workarea buffer
 0E/0F - Address of T/S List sector buffer
 10/11 - Address of data sector buffer
 Output: Byte 07 - File type of file which was OPENed
 0A - Return code (see previous definitions)

CLOSE Write out final sectors, update the Catalog. A CLOSE call is required eventually for every OPEN.

Input: Byte 00 - 02
 0C/0D - Address of file manager workarea buffer
 0E/0F - Address of T/S List sector buffer
 10/11 - Address of data sector buffer
 Output: Byte 0A - Return code

READ Read one or a range of bytes from the file to memory.

WRITE Write one or a range of bytes from memory to the file.

Input: Byte 00 - 03 (READ) 04 (WRITE)
 01 - Subcode:
 00 = No operation
 01 = READ or WRITE one byte only
 02 = READ or WRITE a range of bytes
 03 = POSITION then READ/WRITE one byte
 04 = POSITION then READ/WRITE range

02/03 - (Subcodes 03 or 04) Record number
 04/05 - (Subcodes 03 or 04) Byte offset
 06/07 - (Subcodes 02 or 04) Number of bytes in
 range to be read or written. (Note: for
 WRITE, this length must be one less
 than the actual length to be written)
 08/09 - (Subcodes 02 or 04) Address of range of
 bytes to be written or address of
 buffer to which bytes are to be read.
 08 - (WRITE, Subcodes 01 or 03) Single byte
 to be written.
 0C/0D - Address of file manager workarea buffer
 0E/0F - Address of T/S List sector buffer
 10/11 - Address of data sector buffer
 Output: Byte 02/03 - Record number of current file position
 04/05 - Byte offset of current position in file
 08 - (READ, Subcodes 01 or 03) Byte read
 0A - Return code

DELETE Locate and delete a file, freeing its sectors.
 Input: Byte 00 - 05
 (remainder are the same as with OPEN call type)
 Output: Byte 0A - Return code

CATALOG Produce a catalog listing on the output device.
 Input: Byte 00 - 06
 05 - Drive
 06 - Slot
 0C/0D - Address of file manager workarea buffer
 0E/0F - Address of T/S List sector buffer
 10/11 - Address of data sector buffer
 Output: Byte 0A - Return code

LOCK Lock a file.
 Input: Byte 00 - 07
 (remainder are the same as with OPEN call type)
 Output: Byte 0A - Return code

UNLOCK Unlock a file.
 Input: Byte 00 - 08
 (remainder are the same as with OPEN call type)
 Output: Byte 0A - Return code

RENAME Rename a file.
 Input: Byte 00 - 09
 02/03 - Address of new file name (30 bytes)
 (remainder are the same as with OPEN call type)
 Output: Byte 0A - Return code

POSITION Calculate the location of a record and/or byte offset in the file. Position such that next READ or WRITE will be at that location in the file. A call to POSITION (either explicitly or implicitly using subcodes of READ or WRITE) is required prior to the first READ or WRITE. Bytes 02 through 05 should be set to zeros for a normal position to the beginning of the file.

Input: Byte 00 - 0A

- 02/03 - Relative record number for files with a fixed length record size or zero. First record of file is record 0000.
- 04/05 - Relative byte offset into record or of entire file if record number is zero.
- 0C/0D - Address of file manager workarea buffer.
- 0E/0F - Address of T/S List sector buffer.
- 10/11 - Address of data sector buffer.

Output: Byte 0A - Return code

INIT Initialize a slave diskette. This function formats a diskette and writes a copy of DOS onto tracks 0-2. A VTOC and Catalog are also created. A HELLO program is not stored, however.

Input: Byte 00 - 0B

- 01 - First page of DOS image to be copied to the diskette. Normally \$9D for a 48K machine.
- 04 - Volume number of new diskette.
- 05 - Drive number (01 or 02)
- 06 - Slot number (01-07)
- 0C/0D - Address of file manager workarea buffer.
- 0E/0F - Address of T/S List sector buffer.
- 10/11 - Address of data sector buffer.

Output: Byte 0A - Return code

VERIFY Verify that there are no bad sectors in a file by reading every sector.

Input: Byte 00 - 0C
(remainder are the same as the OPEN call type)

Output: Byte 0A - Return code

DOS BUFFERS

Usually it is desirable to use one of DOS's buffers when calling the file manager to save memory. DOS buffers consist of each of the three buffers used by the file manager (file manager workarea, T/S List sector, and data sector) as well as a 30 byte file name buffer and some link pointers. All together a DOS buffer occupies 595 bytes of memory. The address of the first DOS buffer is stored in the first two bytes of DOS (\$9D00 on a 48K Apple II).

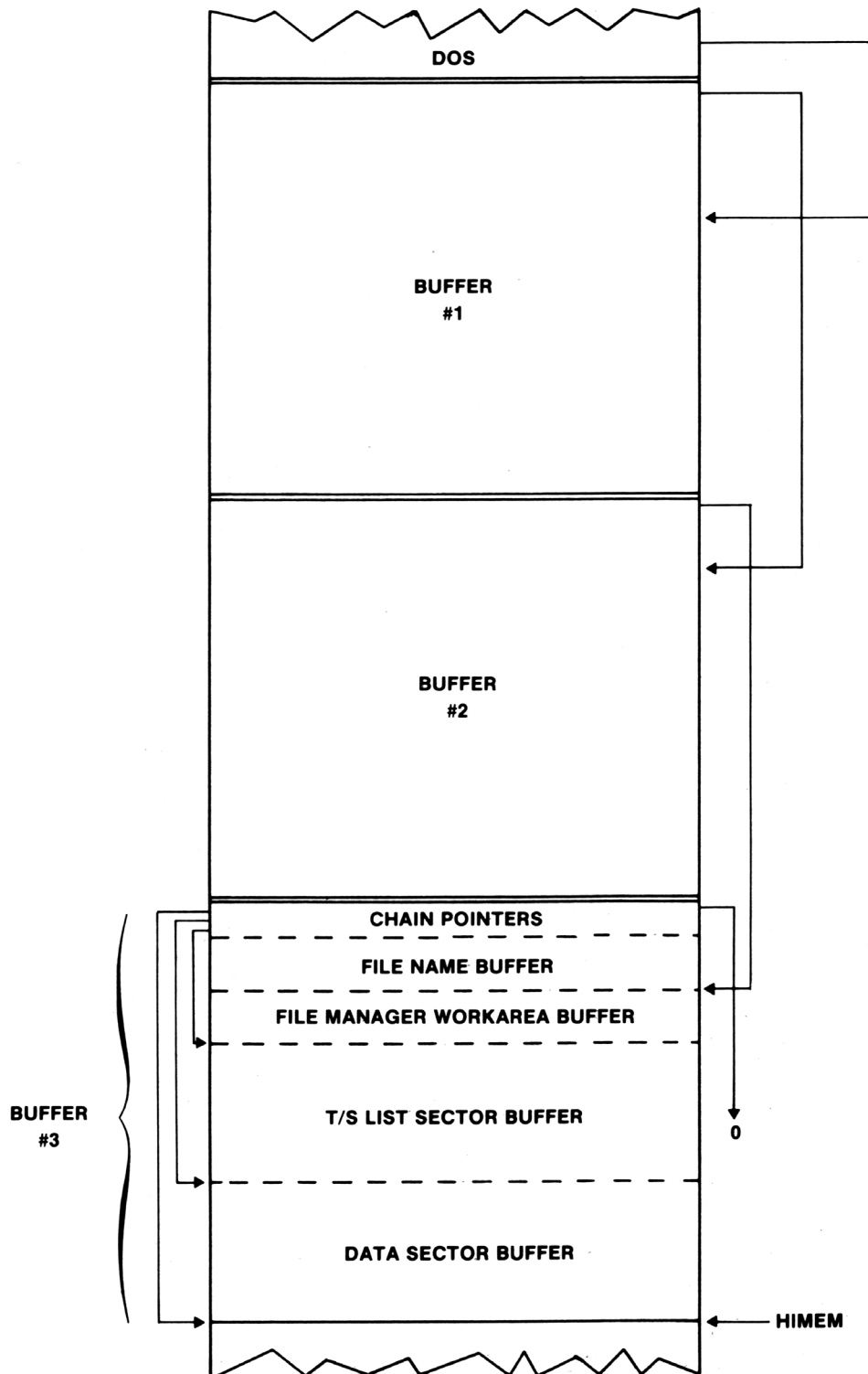


Figure 6.3 DOS File Buffers

The address of the next buffer is stored in the first and so on in a chain of linked elements. The link address to the next buffer in the last buffer is zeros. If the buffer is not being used by DOS, the first byte of the file name field is a hex 00. Otherwise, it contains the first character of the name of the open file. The assembly language programmer should follow these conventions to avoid having DOS reuse the buffer while it's in use. This means that the name of the file should be stored in the buffer to reserve it for exclusive use (or at least a non-zero byte stored on the first character) and later, when the user is through with the buffer, a 00 should be stored on the file name to return it to DOS's use. If the later is not done, DOS will eventually run out of available buffers and will refuse even to do a CATALOG command. A diagram of the DOS buffers for MAXFILES 3 is given in Figure 6.3 and the format of a DOS buffer is given below.

DOS BUFFER FORMAT

BYTE	DESCRIPTION
000/0FF	Data sector buffer (256 bytes in length)
100/1FF	T/S List sector buffer (256 bytes in length)
200/22C	File manager workarea buffer (45 bytes in length)
22D/24A	File name buffer (30 bytes in length)
	First byte indicates whether this DOS buffer is being used. If hex 00, buffer is free for use.
24B/24C	Address (Lo/High) of file manager workarea buffer
24D/24E	Address of T/S List sector buffer
24F/250	Address of data sector buffer
251/252	Address of the file name field of the next buffer on the chain of buffers. If this is the last buffer on the chain then this field contains zeros.

THE FILE MANAGER WORKAREA

The file manager workarea contains the variables which, taken together, constitute all of the information the file manager needs to deal with an open file. Each time the file manager finishes processing a call, it copies all of its important variables into the file manager workarea buffer provided by the caller. Each subsequent time the file manager is called, the first thing it does is to copy the contents



The DOS File Manager

of the file manager workarea buffer back into its variables so that it may resume processing for the file where it left off on the previous call. Ordinarily, the programmer will have no need to worry about the contents of this workarea, since most of the useful information is present in the parameter list anyway. Occasionally, it is handy to know more about the open file. For these cases, the format of the file manager workarea is given below:

FILE MANAGER WORKAREA FORMAT

BYTE	DESCRIPTION
00/01	Track/Sector of first T/S List for file
02/03	Track/Sector of current T/S List for file
04	Flags: 80=T/S List buffer changed and needs writing 40=Data buffer has been changed and needs writing 02=Volume freespace map changed and needs writing
05/06	Track/Sector of current data sector
07	Sector offset into catalog to entry for this file
08	Byte offset into catalog sector to entry for file
09/0A	Maximum data sectors represented by one T/S List
0B/0C	Offset of first sector in current T/S List
0D/0E	Offset of last sector in current T/S List
0F/10	Relative sector number last read
11/12	Sector size in bytes (256)
13/14	Current position in sectors (relative)
15	Current byte offset in this sector
16	Not used
17/18	Fixed record length
19/1A	Current record number
1B/1C	Byte offset into current record
1D/1E	Length of file in sectors
1F	Next sector to allocate on this track
20	Current track being allocated
21/24	Bit map of available sectors on this track (rotated)
25	File type (80=locked) 0,1,2,4=T,I,A,B
26	Slot number times 16 (example: \$60=slot 6)
27	Drive number (01 or 02)
28	Volume number (complemented)
29	Track
2A/2C	Not used

COMMON ALGORITHMS

Given below are several pieces of code which are used when working with DOS:

LOCATE A FREE DOS BUFFER

The following subroutine may be used to locate an unallocated DOS buffer for use with the DOS file manager.

```

FBUFF   LDA    $3D2    LOCATE DOS LOAD POINT
          STA    $1
          LDY    #0
          STY    $0

*
GBUF0   LDA    ($0),Y  LOCATE NEXT DOS BUFFER
          PHA
          INY
          LDA    ($0),Y
          STA    $1
          PLA
          STA    $0
          BNE    GBUF    GOT ONE
          LDA    $1
          BEQ    NBUF    NO BUFFERS FREE

*
GBUF    LDY    #0      GET FILENAME
          LDA    ($0),Y
          BEQ    GOTBUF  ITS FREE
          LDY    #36     ITS NOT FREE
          BNE    GBUF0   GO GET NEXT BUFFER

*
GOTBUF  CLC                INDICATE-GOT A FREE BUFFER
          RTS                RETURN TO CALLER
NBUF    SEC                INDICATE-NO FREE BUFFERS
          RTS                RETURN TO CALLER

```

WHICH VERSION OF DOS IS ACTIVE?

In case the program has version dependent code, a check of the DOS version may be required:

```

CLC
LDA    #0      ADD $16BE TO DOS LOAD POINT
ADC    #$BE
STA    $0
LDA    $3D2

```

```

ADC      #$16
STA      $1
LDY      #0
LDA      ($0),Y  GET DOS VERSION NUMBER (2 OR 3)

```

IS DOS IN THE MACHINE?

The following series of instructions should be used prior to attempting to call RWTS or the file manager to insure that DOS is present on this machine.

```

LDA      $3D0    GET VECTOR JMP
CMP      #$4C    IS IT A JUMP?
BNE      NODOS   NO, DOS NOT LOADED

```

WHICH BASIC IS SELECTED?

Some programs depend upon either the Integer BASIC ROM or the Applesoft ROM. To find out which is active and select the one desired, the following subroutine can be called. First the A register is loaded with a code to indicate which BASIC is desired. \$20 is used for Integer BASIC and \$4C is used for Applesoft. To set up for Applesoft, for example:

```

LDA      #$4C    CODE FOR APPLESOFT
JSR      SETBSC  CALL SUBROUTINE
BNE      ERROR   LANGUAGE NOT AVAILABLE
.
.
.
SETBSC   CMP      $E000  CORRECT BASIC ALREADY THERE?
BEQ      RTS     YES
STA      $C080  NO, SELECT ROM CARD
CMP      $E000  NOW DO WE HAVE IT?
BEQ      RTS     YES
STA      $C081  NO, TRY ROM CARD OUT
CMP      $E000  GOT IT NOW?
RTS      RTS     IN ANY CASE, EXIT TO CALLER

```

SEE IF A BASIC PROGRAM IS IN EXECUTION

To determine if there is a BASIC program running or if BASIC is in immediate command mode, use the following statements:

```

..IF INTEGER BASIC IS ACTIVE...
LDA      $D9
BMI      EXEC    PROGRAM EXECUTING
BPL      NOEXEC  PROGRAM NOT EXECUTING

```

```

..IF APPLESOFT BASIC IS ACTIVE...
    LDX    $76      GET LINE NUMBER
    INX
    BEQ    NOEXEC   PROGRAM NOT EXECUTING
    LDX    $33      GET PROMPT CHARACTER
    CPX    #$DD     PROMPT IS A "]"?
    BEQ    NOEXEC   YES, NOT EXECUTING
    BNE    EXEC     ELSE, PROGRAM IS EXECUTING

```

PROCESS DOS ERROR FROM ASSEMBLY LANGUAGE

If you need to have DOS print out an error message from your assembly language program, use the following piece of code to do it properly:

```

PRERROR    LDA #$F0      Hook output to COUT
            STA $36      instead of DOS
            LDA #$FD      at $FDF0
            STA $37      DOS will put it's own ptr back
            LDY #$0A     See list below of error codes
            LDA ($04),Y  $04 points to FM parmlist
            JMP $A6D2

```

Here is the list of DOS error code offsets:

Error Offset	Text of Error Message
0	RETURN BELL RETURN
1	"LANGUAGE NOT AVAILABLE"
2	"RANGE ERROR" (Bad file manager opcode)
3	"RANGE ERROR" (Bad file manager subcode)
4	"WRITE PROTECTED"
5	"END OF DATA"
6	"FILE NOT FOUND"
7	"VOLUME MISMATCH"
8	"I/O ERROR"
9	"DISK FULL"
10	"FILE LOCKED"
11	"SYNTAX ERROR"
12	"NO BUFFERS AVAILABLE"
13	"FILE TYPE MISMATCH"
14	"PROGRAM TOO LARGE"
15	"NOT DIRECT COMMAND"

Chapter 7

Customizing DOS

Although DOS usually provides most of the functionality needed by the BASIC or assembly language programmer, at times a custom change is required. Making changes to your copy of DOS should only be undertaken when absolutely necessary, since new versions of DOS are released from time to time, and the job of moving several patches to a new version of DOS every few months can become a burden. In addition, wholesale modification of DOS without a clear understanding of the full implications of each change can result in an unreliable system.

SLAVE VS MASTER PATCHING

The usual procedure for making changes to DOS involves “patching” the object or machine language code in DOS. Once a desired change is identified, a few instructions are stored over other instructions within DOS to modify the program. There are three levels at which changes to DOS may be applied.

1. A patch can be made to DOS in memory. If this is done, a later reboot will cause the change to “fall out” or be removed.
2. A patch of the first type can be made permanent by initializing a diskette while running the patched DOS. This procedure creates a slave diskette with a copy of DOS on tracks 0, 1, and 2 which contains the patch. Each time this newly created diskette is booted the patched version of DOS will be loaded. Also, any slave diskettes created by that diskette will also contain the patched version of DOS.
3. The patch is applied directly to a master diskette. This is somewhat more complicated. Either the patch may be made to the image of DOS on the first three tracks of a master diskette using a zap program, or



Custom DOS

MASTER CREATE may be used to write the changed copy of DOS to a new diskette. The following procedure may be followed to do this:

BLOAD MASTER CREATE

Get into the monitor (CALL -151)

Store a \$4C at location \$80D (80D:4C)

Execute MASTER CREATE (8000G)

When MASTER CREATE finishes loading the DOS image it will exit. You may use the monitor to make changes in the image. MASTER CREATE loads DOS into memory at \$1200 such that Boot 2 (RWTS) is loaded first, followed by the main part of DOS starting at \$1C00.

When all patches have been made, reenter MASTER CREATE at location \$82D (82DG).

Complete the MASTER CREATE update normally. The resulting diskette will have the patches applied.

This procedure will work for versions 3.2, 3.2.1, and 3.3 of DOS.

AVOIDING RELOAD OF LANGUAGE CARD

A rather annoying addition to DOS 3.3 was a patch to the Boot 2 code to store a binary zero in the first byte of the Language Card, forcing DOS to reload BASIC (either Integer or Applesoft) for every boot, whether or not the machine was just powered up. When the machine is first powered up this patch is not necessary, since the first byte of the Language Card does not appear to DOS to be either BASIC, and it will reload the card anyway. On subsequent reboots, more often than not, a good copy of BASIC already resides in the Language Card and this patch results in a **LANGUAGE NOT AVAILABLE** error message after booting a slave diskette. Presumably the patch was added for version 3.3 to allow for the eventual possibility that a language like Pascal whose first byte of code just happens to match one of the BASICs would cause strange results in DOS. If the user always powers the machine off and on between using DOS and any other system, the patch may be removed as follows.

At **\$BFD3 (48K)** is a **STA** instruction which stores a zero on the Language Card. This instruction must be made into three no-operation instructions:

BFD3:EA EA EA

A slave diskette may then be INITed using this modified version of DOS and that diskette will have the patch in its DOS. The address of the store instruction for a 32K DOS is \$7FD3 and for a 16K DOS is \$3FD3.

INSERTING A PROGRAM BETWEEN DOS AND ITS BUFFERS

Once in a while it is useful to find a “safe” place to load a machine language program (a printer driver, perhaps) where BASIC and DOS can never walk over it, even if DOS is coldstarted. If the program is less than 200 bytes long, \$300 is a good choice. For larger programs, it is usually better to “tuck” the program in between DOS and its buffers (assuming the program is relocatable and will run at that location). To do this, load the program into low RAM, copy it to high RAM right below \$9D00 (for a 48K machine), over the top of DOS’s buffers, change the first buffer address at \$9D00 to point below your program, (remember to allow 38 extra bytes for the filename and link fields) and JMP to \$3D3 (DOS Coldstart). This will cause DOS to rebuild its buffers below your program and “forget” about the memory your program occupies until the next time DOS is booted. Of course, BASIC can not get at that memory either, since its HIMEM is below the DOS buffers.

You can also do this from an Applesoft program, using the following piece of code:

```
0 POKE 40192, PEEK(40192) - X
1 CALL 42964
```

With X equal to how many pages (256 bytes each) you’d like to reserve.

BRUN OR EXEC THE HELLO FILE

Ordinarily, when DOS finishes booting into memory, it performs a RUN command on the HELLO file in its file name buffer (left there by the INIT command which wrote DOS to the diskette). To change the RUN command to a BRUN or an EXEC, apply the following patch to DOS (48K):

```
9E42:34          (for BRUN)
..or..
9E42:14          (for EXEC)
```

REMOVING THE PAUSE DURING A LONG CATALOG

Normally, when a **CATALOG** command is done on a disk with many files, DOS will pause every time the screen fills with names to allow the user time to see them all. By pressing any key the **CATALOG** continues. If this pause is undesirable, apply the following patch to DOS (48K):

AE34:60

WRITING DOS IN MEMORY TO A DISKETTE

If you'd like to write the current version of DOS onto a disk, use the code below. You might want to make changes to DOS before committing them to a disk, and using this method will allow you to experiment first.

LOAD HELLO

**POKE -21921, 0 : POKE -18448, 0 : POKE -18447, 157 : POKE
-18453, 0 : CALL -18614**

Load the **HELLO** program from the disk so DOS will use it for the startup program. Location **-21921 (43615)** is the "Last Command Executed" byte in DOS. Poking **0** into this location tells DOS that **INIT** was the last command. (**AA5F: 00**)

Locations **-18448 (47088)** and **-18447 (47089)** tells the write routines to start at **\$9D00**, the beginning of DOS in memory. (**B7F0: 00 9D**)

Location **-18453 (47083)** is the Disk Volume Number, and putting **0** into this location will match any disk. (**B7EB: 00**)

The call to **-18614 (46922)** will use DOS's write routine to write DOS onto the current disk using the last slot and drive accessed. (**B74A**)

Chapter 8

DOS Program Logic

This chapter will take a detailed look at the operation of the DOS program itself to aid the Apple user in understanding it and to help make intelligent use of its facilities. Each subroutine and group of variables or constants will be covered separately by storage address. The enterprising programmer may wish to create a disassembly of DOS on the printer and transfer the annotations given here directly to such a listing. Addresses used will be for DOS 3.3 and for a 48K master diskette version of DOS. Slot 6 is assumed. Unless specifically indicated by a \$ character, lengths are given in decimal, addresses in hexadecimal (base 16).

DISK II CONTROLLER CARD ROM - BOOT 0

ADDRESS

C600-C65B This routine is the first code executed when a disk is to be booted. It receives control via PR#6 or C600G or 6 control-P.
Dynamically build a translate table for converting disk codes to six bit hex at location \$356-\$3FF.
Call an RTS instruction in the monitor ROM and extract the return address from the stack to find out the address of this controller card ROM.
Use this address to determine the slot number of this drive by shifting \$Cxxx.
Save the slot number times 16 (\$s0)
Clear disk I/O latches, set read mode, select drive 1, turn disk drive on.
Pull disk arm back over 80 tracks to recalibrate the arm to track zero.
Set up parms to read sector zero on track zero to location \$800.
Execution falls through into a general sector read subroutine at C65C.

C65C-C6FA This subroutine reads the sector number stored at \$3D on the track indicated by \$41 to the address stored at \$26,\$27.
Look for D5/AA/96 sector address header on the disk.
If D5/AA/AD is found and sector data was wanted, go to C6A6.

C683 Handle a sector address block.
Read three double bytes from the disk and combine them to obtain the volume, track, and sector number

of the sector being read from the disk at this time.
Store the track at \$40.
Compare the sector found to the sector wanted and the track found to the track wanted.
If no match, go back to C65C.
Otherwise, if sector is correct, go to C65D to find the sector data itself.
C6A6 Handle sector data block.
Read the 85 bytes of secondary data to \$300-\$355.
Read 256 bytes of primary data to the address stored at \$26,\$27.
Verify that the data checksum is valid.
If not, start over at C65C.
"Nibbilize" the primary and secondary data together into the primary data buffer (\$26,\$27).
Increment \$27 (address page of read data) and \$3D (sector number to be read) and check against \$800 to see if additional sectors need to be read.
If so, reload slot*16 and go back to C65C to read next sector. (This feature is not used when loading DOS but is used when loading from a BASICS diskette.)
Otherwise, go to \$801 to begin executing the second stage of the bootstrap.

FIRST RAM BOOTSTRAP LOADER - BOOT 1

ADDRESS

0801-084C This routine loads the second RAM loader, Boot 2, including RWTS, into memory and jumps to it.
If this is not the first entry to Boot 1, go to \$81F.
Get slot*16 and shift down to slot number.
Create the address of the ROM sector read subroutine (C65C in our case) and store it at \$3E,\$3F.
Pick up the first memory page in which to read Boot 2 from location \$8FE, add the length of Boot 2 in sectors from \$8FF, and set that value as the first address to which to read (read last page first).
081F Get sector to read, if zero, go to \$839.
Translate theoretical sector number into physical sector number by indexing into skewing table at \$84D.
Decrement theoretical sector number (8FF) for next iteration through.
Set up parameters for ROM subroutine (C65C) and jump to it. It will return to \$801 when the sector has been read.
0839 Adjust page number at 8FE to locate entry point of Boot 2.
Perform a PR#0 and IN#0 by calling the monitor.
Initialize the monitor (TEXT mode, standard window, etc.)
Get slot*16 again and go to Boot 2 (\$3700 for a master disk, \$B700 in its final relocated location).

DOS 3.3 MAIN ROUTINES

ADDRESS

9D00-9D0F Relocatable address constants

- 9D00 Address of first DOS buffer at its file name field.
- 9D02 Address of the DOS keyboard intercept routine.
- 9D04 Address of the DOS video intercept routine.
- 9D06 Address of the primary file name buffer.
- 9D08 Address of the secondary (RENAME) file name buffer.
- 9D0A Address of the range length parameter used for LOAD.
- 9D0C Address of the DOS load address (\$9D00).
- 9D0E Address of the file manager parameter list.

9D10-9D1C DOS video (CSWL) intercept's state handler address table. States are used to drive the handling of DOS commands as they appear as output of PRINT statements and this table contains the address of the routine which handles each state from state 0 to state 6.

9D1E-9D55 Command handler entry point table. This table contains the address of a command handler subroutine for each DOS command in the following standard order:

INIT	A54F
LOAD	A413
SAVE	A397
RUN	A4D1
CHAIN	A4F0
DELETE	A263
LOCK	A271
UNLOCK	A275
CLOSE	A2EA
READ	A51B
EXEC	A5C6
WRITE	A510
POSITION	A5DD
OPEN	A2A3
APPEND	A298
RENAME	A281
CATALOG	A56E
MON	A233
NOMON	A23D
PR#	A229
IN#	A22E
MAXFILES	A251
FP	A57A
INT	A59E
BSAVE	A331
BLOAD	A35D
BRUN	A38E
VERIFY	A27D

9D56-9D61 Active BASIC entry point vector table. The addresses stored here are maintained by DOS such that they apply to the current version of BASIC running.

- 9D56 Address of CHAIN entry point to BASIC.
- 9D58 Address of RUN.
- 9D5A Address of error handler.
- 9D5C Address of BASIC coldstart.
- 9D5E Address of BASIC warmstart.
- 9D60 Address of BASIC relocate (APPLESOFT only).

9D62-9D6B Image of the entry point vector for INTEGER BASIC. This image is copied to 9D56 if INTEGER BASIC is made

active.

9D6C-9D77 Image of the entry point vector for the ROM version of APPLESOFT.

9D78-9D83 Image of the entry point vector for the RAM version of APPLESOFT.

9D84-9DBE DOS coldstart entry routine.
Get the slot and drive numbers and store as default values for command keywords.
Copy APPLESOFT ROM or INTEGER BASIC entry point vector into current BASIC entry point vector.
Remember which BASIC is active.
Go to 9DD1.

9DBF-9DE9 DOS warmstart entry routine.
Get the remembered BASIC type and set the ROM card as necessary (calls A5B2).

9DD1 Remember whether entry is coldstart or warmstart
Call A851 to replace DOS keyboard and video intercepts.
Set NOMON C,I,O.
Set video intercept handler state to 0.
Coldstart or warmstart the current BASIC (exit DOS).
(DOS will next gain control when BASIC prints its input prompt character)

9DEA-9E50 First entry processing for DOS. This routine is called by the keyboard intercept handler when the first keyboard input request is made by BASIC after a DOS coldstart.
If RAM APPLESOFT is active, copy its entry point vector to the active BASIC entry point vector and blank out the primary file name buffer so that no HELLO file will be run.
Set MAXFILES to 3 by default.
Call A7D4 to build the DOS file buffers.
If an EXEC was active, close the EXEC file
Set the video intercept state to 0 and indicate warmstart status by calling A75B.
If the last command executed was not INIT (this DOS was not just booted), go to 9E45.
Otherwise, copy an image of the DOS jump vector to \$3D0-\$3FF.
Point \$3F2,\$3F3 to DOS warmstart routine.
Set the AUTOSTART ROM power-up byte since the RESET handler address was changed.
Set the command index for RUN (to run the HELLO file) and go to A180 to execute it.

9E45 See if there is a pending command.
If so, go to A180 to execute it. Otherwise, return to caller.

9E51-9E7F An image of the DOS 3-page jump vector which the above routine copies to \$3D0-\$3FF. See Chapter 5 for a description of its contents.

9E81-9EB9 DOS keyboard intercept routine.
Call 9ED1 to save the registers at entry to DOS.
If not coldstarting or reading a disk file, go to 9E9E.
Get value in A register at entry and echo it on the screen (erases flashing cursor).
If in read state (reading a file) go to A626 to get next byte from disk file.

Otherwise, call 9DEA to do first entry processing.
 Put cursor on screen in next position.
 If EXECing, call A682 to get the next byte from the EXEC file.
 Set the video intercept state to 3 (input echo).
 Call 9FBA to restore the registers at entry to DOS.
 Call the true keyboard input routine.
 Save the input character so that it will be restored with the registers in the A register.
 Do the same with the new X register value.
 Exit DOS via 9FB3.

9EBA-9EBC A jump to the true KSWL handler routine.

9EBD-9ED0 DOS video intercept routine.
 Call 9ED1 to save the registers at entry to DOS.
 Get the video intercept state and, using it as an index into the state handler table (9D11), go to the proper handler routine, passing it the character being printed.

9ED1-9EEA Common intercept save registers routine.
 Save the A, X, Y, and S registers at AA59-AA5C.
 While in DOS, restore the true I/O handlers (KSWL and CSWL) to \$36-\$39.
 Return to caller.

9EEB-9F11 State 0 output handler. --start of line--
 If a RUN command was interrupted (by loading RAM APPLESOFTE) go to 9F78 to complete it.
 If read flag is on (file being read) and output is a "?" character (BASIC INPUT), go to state 6 to skip it.
 If read flag is on and output is prompt character (\$33) go to state 2 to ignore the line.
 Set state to 2 (ignore non-DOS command) just in case.
 If output character is not a control-D, go to state 2.
 Otherwise, set state to 1 (collect possible DOS command), set line index to zero, and fall through to state 1.

9F12-9F22 State 1 output handler. --collect DOS command--
 Using line index, store character in input buffer at \$200.
 Increment line index.
 If character is not a carriage return, exit DOS via 9F95 (echo character on screen if MON I).
 Otherwise, go to command scanner at 9FCD.

9F23-9F2E State 2 output handler. --non-DOS command ignore--
 If the character is not a carriage return, exit DOS via 9FA4 (echo character on screen).
 Otherwise, set state back to 0 and exit DOS via 9FA4.

9F2F-9F51 State 3 output handler. --INPUT statement handler--
 Set state to 0 in case INPUT ends.
 If character is not a carriage return, echo it on screen as long as EXEC is not in effect with NOMON I but exit DOS in any case. (KSWL will set state=3)
 Otherwise, call A65E to see if BASIC is executing a program or is in immediate mode. If EXEC is running or if BASIC is in immediate mode, go to state 1 to collect the possible DOS command.
 Otherwise, exit DOS, echoing the character as

appropriate.

9F52-9F60 State 4 output handler. --WRITE data to a file--
If the character is a carriage return, set state to 5 (start of write data line).
Call A60E to write the byte to the disk file.
Exit DOS with echo on screen if MON 0.

9F61-9F70 State 5 output handler. --Start of WRITE data line--
If the character is a control-D, go to state 0 to immediately exit write mode.
If the character is a line feed, write it and exit, staying in state 5.
Otherwise, set the state to 4 and go to state 4.

9F71-9F77 State 6 output handler. --Skip prompt character--
Set state to 0.
Exit DOS via 9F9D (echo if MON I).

9F78-9F82 Finish RUN command, interrupted by APPLESOFT RAM LOAD
Reset the "RUN interrupted" flag.
Call A851 to replace the DOS CSWL/KSWL intercepts.
Go to A4DC to complete the RUN command.

9F83-9F94 DOS command scanner exit to BASIC routine.
If first character of command line is control-D, go to echo exit (9F95).
Otherwise, set things up so BASIC won't see the DOS command by passing a zero length line (only a carriage return). Fall through to echo exit.

9F95-9FB0 Echo character on screen (conditionally) and exit DOS
9F95 Echo only if MON C set, otherwise, go to 9FB3.
9F99 Echo only if MON 0 set, otherwise, go to 9FB3.
9F9D Echo only if MON I set, otherwise, go to 9FB3.
9FA4 Always echo character.
Call 9FBA to restore registers at entry to DOS.
Call 9FC5 to echo character on screen.
Save contents of the registers after echoing.
Fall through to DOS exit routine.

9FB3-9FC4 DOS exit routine and register restore.
Call A851 to put back DOS KSWL/CSWL intercepts.
Restore S (stack) register from entry to DOS.

9FBA DOS register restore subroutine.
Restore registers from first entry to DOS and return to caller.

9FC5-9FC7 A jump to the true CSWL routine.

9FC8-9FCC Skip a line on the screen.
Load a carriage return into the A register and call 9FC5 to print it.

9FCD-A179 DOS command parse routine.
Set the command index to -1 (none).
Reset the pending command flag (none pending).

9FD6 Add one to command index.
If first character is a control-D, skip it.
Flush to a non-blank (call A1A4).
Compare command to command name in command name table at A884 for the current command index.
If it doesn't match and if there are more entries left to check, go back to 9FD6.
If it does match, go to A01B.
Otherwise, if command was not found in the table, check to see if the first character was a control-D.
If so, go to A6C4 to print "SYNTAX ERROR".
Otherwise, call A75B to reset the state and warmstart

flag and go to 9F95 to echo the command and exit.
 (the command must be for BASIC, not DOS)

A01B Compute an index into the operand table for the command which was entered.
 Call A65E to see if a BASIC program is executing.
 If not, and the command is not a direct type command, (according to the operand table) go to A6D2 to print "NOT DIRECT COMMAND".
 Otherwise, if the command is RUN, make the prompt character (\$33) non-printing.
 Check the operand table to see if a first filename is a legal operand for this command.
 If not, go to A0A0.
 Otherwise, clear the filename buffer (call A095).
 Flush to the next non-blank (call A1A4) and copy the filename operand to the first filename buffer.
 Skip forward to a comma if one was not found yet.
 If a second filename is legal for this command, use the code above to copy it into the second filename buffer.
 Check both filenames to see if they are blank.
 If one was required by the command but not given, give a syntax error or pass it through to BASIC.
 (As in the case of LOAD with no operands)
 If all is well, go to A0D1 to continue.

A095 A subroutine to blank both filename buffers.

A0A0 Indicate no filename parsed.
 Check operand table to see if a positional operand is expected.
 If not, go to A0D1 to continue.
 Otherwise, call A1B9 to convert the numeric operand.
 If omitted, give syntax error.
 If number converted exceeds 16, give "RANGE ERROR"
 If number is supposed to be a slot number, give "RANGE ERROR" if it exceeds 7.
 If number is not a slot number, give "RANGE ERROR" if it is zero. (MAXFILES 0 is a no-no)

A0D1 Set defaults for the keyword operands (V=0,L=0,B=0)

A0E8 Get the line offset index and flush to the next non-blank, skipping any commas found.
 If we are not yet to the end of the line, go to A10C.
 Check to see if any keywords were given which were not allowed for this command.
 If not, go to A17A to process the command.

A10C Lookup the keyword found on the command line in the table of valid keywords (A940)
 If not in table, give "SYNTAX ERROR" message.
 Get its bit position in the keywords-given flag.
 If the keyword does not have an operand value, go to A164.
 Otherwise, indicate keyword found in flag.
 Convert the numeric value associated with keyword.
 Give "SYNTAX ERROR" message if invalid.
 Check to see if the number is within the acceptable range as given in the keyword valid range table at A955.
 Save the value of the keyword in the keyword values table starting at AA66.
 Go parse the next keyword. Go to A0E8.

A164 Indicate C, I, or O keywords were parsed.
Update the MON value in the keyword value table appropriately.
Go parse the next keyword. Go to A0E8.

A17A-A17F Call A180 to process the command, then exit via echo at 9F83.

A180-A192 Do command.
Reset the video intercept state to zero.
Clear the file manager parameter list.
Using the command index, get the address of the command handling routine from the command handler routine table at 9D1E and go to it.
Command handler will exit to caller of this routine.

A193-A1A3 Get next character on command line and check to see if it is a carriage return or a comma.

A1A4-A1AD Flush command line characters until a non-blank is found.

A1AE-A1B8 Clear the file manager parameter list at B5BA to zeros.

A1B9-A1D5 Convert numeric operand from command line. Call either A1D6 (decimal convert) or A203 (hex convert) depending upon the presence or lack thereof of a dollar sign (\$).

A1D6-A202 Decimal convert subroutine.

A203-A228 Hexadecimal convert subroutine.

A229-A22D PR#n command handler.
Load the parsed numeric value and exit via FE95 in the monitor ROM.

A22E-A232 IN#n command handler.
Load the parsed numeric value and exit via FE8B in the monitor ROM.

A233-A23C MON command handler.
Add new MON flags to old in AA5E and exit.

A23D-A250 NOMON command handler.
If C was given, put out a carriage return since this line was echoed but its CR was not.
Turn off the proper bits in AA5E and exit.

A251-A262 MAXFILES command handler.
Turn off any EXEC file which is active.
Close all open files (call A316).
Set the new MAXFILES number at AA57.
Go to A7D4 to rebuild the DOS file buffers and exit.

A263-A270 DELETE command handler.
Load the delete file manager opcode (05).
Call the file manager open driver (A2AA) to perform the delete.
Find the file buffer used to do the delete and free it (call A764).
Exit to caller.

A271-A274 LOCK command handler.
Load the lock file manager opcode (07) and go to A277.

A275-A27C UNLOCK command handler.
Load the unlock file manager opcode (08).

A277 Call the file manager open driver (A2AA) to perform the desired function.
Exit to the caller via close (A2EA).

A27D-A280 VERIFY command handler.
Load the verify file manager opcode (0C) and go to

```

A277 to perform function.
A281-A297 RENAME command handler.
    Store address of second file name in file manager
    parameter list.
    Load the rename file manager opcode (09).
    Call the file manager driver at A2C8.
    Exit via close (A2EA).
A298-A2A2 APPEND command handler.
    Call A2A3 to OPEN the file.
    Read the file byte by byte until a zero is found.
    If append flag is on, add one to record number
    and turn flag off.
    Exit via a call to POSITION.
A2A3-A2A7 OPEN command handler.
    Set file type as TEXT.
    Go to A3D5 to open file.
A2A8-A2E9 Command handler common file management code.
    Set opcode to OPEN.
    A2AA If no L value was given on the command, use 0001 and
    store record length value in file manager parmlist.
    A2C8 Close file if already open.
    Is there an available file buffer?
    If not, issue "NO FILE BUFFERS AVAILABLE" message.
    Point $40,$41 at the free file buffer.
    Copy filename to file buffer (allocates the buffer)
    (A743).
    Copy buffer pointers to file manager parmlist (A74E).
    Finish filling in the file manager parmlist (A71A).
    Set operation code in parmlist.
    Exit through the file manager driver.
A2EA-A2FB CLOSE command handler.
    If no filename was given as part of command,
    go to A316 to close all files.
    Otherwise, find the open file buffer for filename
    (A764).
    If no such file open, exit to caller.
    Otherwise, close file and free buffer (A2FC).
    Go back through CLOSE command handler to make sure
    there are not more open buffers for the same file.
A2FC-A315 Close a file and free its file buffer.
    Find out if this buffer is EXEC's (A7AF).
    If so, turn EXEC flag off.
    Release the buffer by storing a $00 on its filename
    field.
    Copy file buffer pointers to the file manager
    parmlist.
    Set file manager opcode to CLOSE.
    Exit through the file manager driver routine.
A316-A330 Close all open files.
    Point to first file buffer (A792).
    Go to A320.
    A31B Point to next file buffer on chain (A79A).
    If at end of chain, exit to caller.
    A320 Is this file buffer EXEC's?
    If so, skip it and go to A31B.
    Is it not in use (open)?
    If so, skip it and go to A31B.
    Otherwise, close it and free it (A2FC).
    Go to A316 to start all over.

```

A331-A35C BSAVE command handler.
Insure that the A and L keywords were present on the command.
If not, issue "SYNTAX ERROR" message.
Open and verify a B type file (A3D5).
Write the A keyword value as the first two bytes of the file.
Write the L keyword value as the next two bytes of the file.
Use the A value to exit by writing a range of bytes from memory to the file.

A35D-A38D BLOAD command handler.
Open the file, ignoring its type.
Insure the type is B.
If not, issue "FILE TYPE MISMATCH" message.
Otherwise, open B type file and test file type (A3D5).
Read the A value from the first two bytes of file.
If A keyword was not given, use the value just read.
Read L value as next two bytes in file.
Go to A471 to read range of bytes to memory from file

A38E-A396 BRUN command handler.
Call BLOAD command handler to load file into memory.
Replace DOS intercepts.
Exit DOS by jumping to the A address value to begin execution of the binary program.

A397-A3D4 SAVE command handler.
Get the active BASIC type (AAB6).
If INTEGER, go to A3BC.
If APPLESOFT, test \$D6 flag to see if program is protected.
If so, issue "PROGRAM TOO LARGE" message.
Otherwise, open and test for A type file (A3D5).
Compute program length (PGMEND-LOMEM).
Write this two byte length to file.
Exit by writing program image from LOMEM as a range of bytes (A3FF).

A3BC Open and test for I type file (A3D5).
Compute program length (HIMEM-PGMSTART).
Write this two byte length to file.
Exit by writing program image from PGMSTART as a range of bytes (A3FF).

A3D5-A3DF Open and test file type.
Set file type wanted in file manager parmlist.
Call A2A8 to open file.
Go to A7C4 to check file type.

A3E0-A3FE Write a 2 byte value to the open file.
Store value to be written in file manager parmlist.
Set write one byte opcodes.
Call file manager driver.
Call it again to write second byte and exit to caller.

A3FF-A40F Read/write a range of bytes.
Set the address of the range in file manager parmlist.
Set subcode to read or write a range of bytes.
Call the file manager driver.
Close the file.
Exit through the VERIFY command handler to insure data was written ok.

A410-A412 Issue "FILE TYPE MISMATCH" message.

A413-A479 LOAD command handler.

```

Close all files (A316).
Open the file in question.
Is it an A or I type file?
If not, issue "FILE TYPE MISMATCH" message.
Which BASIC is active?
If INTEGER, go to A450.
Select APPLESOFT BASIC (A4B1). This call could result
in DOS losing control if the RAM version must be
run.
Read first two bytes of file as length of program.
Add length to LOMEM (program start) to compute
program end.
Is program end beyond HIMEM?
If so, close file and issue "PROGRAM TOO LARGE".
Set program end and start of variables pointers.
Read program as range of bytes to program start.
Replace DOS intercepts (A851).
Go to BASIC's relocation routine to convert a RAM
APPLESOFT program to ROM and vice versa as needed.
A450 Select INTEGER BASIC (A4B1).
Read length of program (first two bytes in file).
Compute program start (HIMEM-LENGTH).
If zero or less than LOMEM, issue "PROGRAM TOO LARGE"
message and close file.
Set program start pointers.
Read program into memory as a range of bytes.
Exit to caller.
A47A-A4AA Read two bytes from file (Address or Length).
Set up parmlist to read two bytes to range length
field (AA60).
Call file manager driver.
Store value read as range length in file manager
parmlist just in case it was a length.
A4AB-A4B0 Close file and issue "PROGRAM TOO LARGE" message.
A4B1-A4D0 Select desired BASIC.
If desired BASIC is already active, exit to caller.
Save current command index in case we must RUN
APPLESOFT.
If INTEGER, go to A59E to select it.
Otherwise, copy primary file name to secondary
buffer to save it in case RAM APPLESOFT is needed.
Go to A57A to set APPLESOFT.
A4D1-A4E4 RUN command handler.
If APPLESOFT is active, set RUN intercepted flag so
that RUN can complete after APPLESOFT is loaded.
Call LOAD command handler to load the program.
Skip a line on the screen.
Put DOS intercepts back.
Go to the RUN entry point in the current BASIC.
A4E5-A4EF INTEGER BASIC RUN entry point intercept.
Delete all variables (CLR equivalent).
Go to the CHAIN entry point in INTEGER BASIC.
A4F0-A4FB CHAIN command handler.
Call the LOAD command handler to load the program.
Skip a line.
Replace DOS intercepts.
Go to current BASIC's CHAIN entry point.
A4FC-A505 APPLESOFT ROM RUN entry point intercept.
Call APPLESOFT to clear variables.

```

Reset ONERR.
Go to RUN entry point.

A506-A50D APPLESOFT RAM RUN entry point intercept.
Call APPLESOFT to clear variables.
Reset ONERR.
Go to RUN entry point.

A510-A51A WRITE command handler.
Call READ/WRITE common code (A526).
Set CSWL state to 5 (WRITE mode line start).
Exit DOS (9F83).

A51B-A525 READ command handler.
Call READ/WRITE common code (A526).
Set READ mode flag in status flags (AA51).
Exit DOS (9F83).

A526-A54E READ/WRITE common code.
Locate the open file buffer for this file (A764).
If not open, open it.
Copy file buffer addresses to file manager parmlist.
If R or B were given on command, copy to parmlist
and issue a POSITION call to file manager.
Exit to caller.

A54F-A56D INIT command handler.
If V was given, use it. Otherwise, use 254.
Store first page number of DOS in file manager
parmlist.
Call file manager driver to INIT diskette.
Exit through SAVE to store greeting program on disk.

A56E-A579 CATALOG command handler.
Call file manager with CATALOG opcode.
Set new V value as default for future commands.
Exit to caller.

A57A-A59D FP command handler.
Set ROM card, if any, for APPLESOFT (A5B2).
If successful, coldstart DOS (9D84).
Otherwise, set status flag to indicate INTEGER BASIC
is active.
Set primary filename buffer to "APPLESOFT".
Set flags to indicate RAM APPLESOFT and coldstart.
Go to RUN command handler.

A59E-A5B1 INT command handler.
Set ROM card, if any, for INTEGER BASIC (A5B2).
If not successful, issue "LANGUAGE NOT AVAILABLE".
Otherwise, clear RUN intercepted flag.
Coldstart DOS (9D84).

A5B2-A5C5 Set ROM to desired BASIC.
(This routine is passed a \$4C for APPLESOFT or a \$20
for INTEGER, since these bytes appear at \$E000 in
these BASICS. It will work regardless of which
BASIC is onboard.)
If desired BASIC is already available, exit.
Try selecting ROM card.
If desired BASIC is now available, exit.
Try selecting onboard ROM.
If desired BASIC is now available, exit.
Otherwise, exit with error return code.

A5C6-A5DC EXEC command handler.
Open the file (A2A3).

Copy file buffer address to EXEC's buffer pointer at AAB4,AAB5.
Set EXEC active flag (AAB3).
Jump into POSITION command handler to skip R lines.

A5DD-A60D POSITION command handler.
Locate the open file buffer (A764).
If not found, open one as a TEXT file.
Copy buffer pointers to file manager parmlist.
If R was not given on command, exit.

A5F2 Otherwise, test R value for zero and exit if so.
Decrement R value by one.
Read file byte by byte until a carriage return (end of line - \$8D) is reached.
If at end of file, issue "END OF FILE" message.
Otherwise, go to A5F2 to skip next record.

A60E-A625 Write one data byte to file.
Insure that BASIC is running a program (A65E).
If not, close file and warmstart DOS.
Set up file manager parmlist to write the data byte to the open file.
Call file manager and exit.

A626-A65B Read one data byte from file.
Insure that BASIC is running a program (A65E).
If not, close file and warmstart DOS.
Set CSWL intercept state to 6 (skip prompt character).

A630 Read next file byte (A68C).
If not at end of file, go to A644.
Otherwise, close file.
If state is not 3 (EXEC) issue "END OF DATA" message.
Exit to caller.

A644 If data byte is lower case character, turn its most significant bit off to fool GETIN routine in monitor.
Store data byte in A register saved at entry to DOS.
Using line index, turn high bit back on in previous data byte stored at \$200 (input line buffer) to make it lower case if necessary.
Exit DOS (9FB3).

A65E-A678 Test to see if BASIC is running a program or is in immediate command mode.
If active BASIC is INTEGER, go to A672.
If line number is greater than 65280 and prompt is "]" then APPLESOFT is in immediate mode.
Otherwise, it is executing a program.
Exit to caller with appropriate return code.

A672 Check \$D9 to determine whether BASIC is executing a program and exit with proper return code.

A679-A681 Close current file and warmstart DOS.

A682-A68B EXEC read one byte from file.
Select EXEC file buffer.
Copy file buffer addresses to file manager parmlist.
Set state to 3 (input echo).
Go to A62D to read a file byte.

A68C-A69C Read next text file byte.
Set up file manager parmlist to read one byte.
Call file manager driver.
Return to caller with the data byte.

A69D-A6A7 Set \$40,\$41 to point to EXEC file buffer.

A6A8-A6C3 File manager driver routine.
Call the file manager itself (AB06).

If no errors, exit to caller.
Otherwise, point \$40,\$41 at file buffer.
If found, release it by storing a zero on the file name field.
If error was not "END OF DATA", print error message.
Otherwise, pretend a \$00 was read and return to caller.

A6C4-A6D4 Miscellaneous error messages.
A6C4 "COMMAND SYNTAX ERROR"
A6C8 "NO FILE BUFFERS AVAILABLE"
A6CC "PROGRAM TOO LARGE"
A6D0 "FILE TYPE MISMATCH"

A6D5-A701 Error handler.
Set warmstart flag and clear status (BFE6).
If APPLESOFT ONERR is active, go to A6EF.
Otherwise, print RETURN BELL RETURN.
Print text of error message (A702).
Print another RETURN.

A6EF Replace DOS intercepts.
If a BASIC program is in execution, pass error code to BASIC's error handler.
Otherwise, warmstart BASIC.

A702-A719 Print text of error message.
Using the error number as an index, print the message text from the message table (A971) byte by byte.
Last character has most significant bit on.

A71A-A742 Complete file manager parameter list.
Copy Volume value to parmlist.
Copy Drive value to parmlist.
Copy Slot value to parmlist.
Copy address of primary filename buffer to parmlist.
Save file buffer address in \$40,\$41.
Return to caller.

A743-A74D Copy primary filename to file buffer filename field.

A74E-A75A Copy current buffer pointers to file manager parmlist
Copy file manager workarea buffer pointer.
Copy T/S List sector buffer pointer.
Copy data sector buffer address.
Copy next file buffer link address.
Return to caller.

A75B-A763 Reset state to 0 and set warmstart flag.

A764-A791 Locate an open or free file buffer.
Assume there are no free file buffers by zeroing \$45.
Point \$40,\$41 at first buffer on chain.
Go to A773.

A76E Point \$40,\$41 at next buffer on chain.
If at end of chain, exit with file not open code.

A773 Get first byte of filename field.
If zero (file buffer free), save file buffer address at \$44,\$45 as an available buffer and go to A76E.
Otherwise, see if name in primary filename buffer matches the name in this file buffer.
If not, go to A76E to get next buffer.
If so, return to caller with open file found code.

A792-A799 Point \$40,\$41 at first file buffer on chain.

A79A-A7A9 Point \$40,\$41 at next file buffer on chain.

A7AA-A7AE Get first byte of file name in file buffer.

A7AF-A7C3 See if current buffer belongs to EXEC.
Is EXEC active?

```

        If not, exit.
        If so, does current buffer address match EXEC's?
        Return to caller with appropriate code.
A7C4-A7D3 Check file type.
        Does file type of open file match desired file type?
        If so, exit.
        Otherwise, turn lock bit off and test again.
        If ok, exit.
        Otherwise, close file and issue "FILE TYPE MISMATCH".
A7D4-A850 Initialize (build) DOS file buffer chain.
        Set $40,$41 to point to first buffer.
        Set counter to MAXFILES value.
A7E5 Store zero on filename field to mark as free.
        Set up link pointers in buffer to point to file
        manager workarea (45 bytes prior to filename field).
        Set up link pointer to T/S List sector buffer (-256
        bytes from file manager workarea buffer).
        Set up link pointer to data sector buffer 256 bytes
        before that.
        Decrement counter.
        If zero, go to A82D to set HIMEM.
        Otherwise, set link to next file buffer as 38 bytes
        prior to data sector buffer.
        Go to A7E5 to set up next buffer.
A82D Set link of last buffer to $0000.
        If INTEGER BASIC is active, go to A846.
        Otherwise, set APPLESOFT's HIMEM and STRING START
        pointers in zeropage to point just below the last
        buffer.
        Exit to caller.
A846 Set INTEGER BASIC's HIMEM and PROGRAM START pointers
        to point just below the last buffer.
        Exit to caller.
A851-A883 Replace DOS keyboard/video intercept vectors.
        Is DOS keyboard (KSWL) vector still set?
        If so, go to A86A.
        Otherwise, save current KSWL vector ($38,$39) at
        AA55,AA56 and replace with DOS intercept routine's
        address.
A86A Is DOS video (CSWL) vector still set?
        If so, exit to caller.
        Otherwise, save current CSWL vector ($36,$37) at
        AA53,AA54 and replace with DOS intercept routine's
        address.
        Exit to caller.
A884-A908 DOS command name text table.
        This table consists of the ASCII name for each DOS
        command in order of command index values, with the
        last character of each indicated by the MSB being
        on. Commands in order are:
            INIT,LOAD,SAVE,RUN,CHAIN,DELETE,LOCK,UNLOCK,CLOSE,
            READ,EXEC,WRITE,POSITION,OPEN,APPEND,RENAME,
            CATALOG,MON,NOMON,PR#,IN#,MAXFILES,FP,INT,BSAVE,
            BLOAD,BRUN,VERIFY.
        Example: INIT is $49 $4E $49 $D4 ( I N I T )
A909-A940 Command valid keywords table.
        This table is used to determine which keywords are
        required or may be given for any DOS command.
        Each command has a two byte entry with 16 flags,

```

indicating which keywords may be given. The flag bit settings are as follows:

BIT	MEANING
0	Filename legal but optional
1	Command has no positional operand
2	Filename #1 expected
3	Filename #2 expected
4	Slot number positional operand expected
5	MAXFILES value expected as positional operand
6	Command may only be issued from within a program
7	Command may create a new file if file not found
8	C, I, O keywords legal
9	V keyword legal
10	D keyword legal
11	S keyword legal
12	L keyword legal
13	R keyword legal
14	B keyword legal
15	A keyword legal

Thus, for a typical command, OPEN, where the value is \$2378, bits 2, 6, 7, 9, 10, 11, and 12 are set so the command has one filename operand, may only be issued from within a program, may create a new file, and the V, D, S, and L keywords are legal.

The command entries are:

INIT	2170
LOAD	A070
SAVE	A170
RUN	A070
CHAIN	2070
DELETE	2070
LOCK	2070
UNLOCK	2070
CLOSE	6000
READ	2206
EXEC	2074
WRITE	2206
POSITION	2204
OPEN	2378
APPEND	2270
RENAME	3070
CATALOG	4070
MON	4080
NOMON	4080
PR#	0800
IN#	0800
MAXFILES	0400
FP	4070
INT	4000
BSAVE	2179
BLOAD	2071
BRUN	2071
VERIFY	2070

A941-A94A Keyword name table.

This table contains all the ASCII names of the DOS keywords in standard order. Each keyword name occupies one byte:

V,D,S,L,R,B,A,C,I,O

A94B-A954 Keyword flag bit positions table.

This table gives the bit positions for each keyword into the second byte of the command valid keyword table above and in the flag (AA65) which indicates which keywords were present on the command line.

The bit positions are:

V - 40
 D - 20
 S - 10
 L - 08
 R - 04
 B - 02
 A - 01
 C - C0 ...
 I - A0 ... not used in valid keyword table
 O - 90 ...

A955-A970 Keyword value valid range table.

This table indicates the range any keyword value may legally have. Each keyword has a four byte entry, two bytes of minimum value, and two bytes of maximum value. Values are:

KEYWORD	MIN	MAX
V	0	254
D	1	2
S	1	7
L	1	32767
R	0	32767
B	0	32767
A	0	65535

C, I, and O do not appear in this table since they do not have numeric values.

A971-AA3E Error message text table.

This table contains the text for each error code in order of error code number:

NUMBER	TEXT
0	RETURN BELL RETURN
1	"LANGUAGE NOT AVAILABLE"
2	"RANGE ERROR" (Bad file manager opcode)
3	"RANGE ERROR" (Bad file manager subcode)
4	"WRITE PROTECTED"
5	"END OF DATA"
6	"FILE NOT FOUND"
7	"VOLUME MISMATCH"
8	"I/O ERROR"
9	"DISK FULL"
10	"FILE LOCKED"
11	"SYNTAX ERROR"
12	"NO BUFFERS AVAILABLE"
13	"FILE TYPE MISMATCH"
14	"PROGRAM TOO LARGE"
15	"NOT DIRECT COMMAND"

AA3F-AA4F Error message text offset index table.

This table contains the offset in bytes to the text of any given error message in the table above.

Entries are one byte each for each error code number.

AA4F-AA65 DOS main routines variables.

AA4F Current file buffer address (2 bytes).

AA51 Status flags: \$01=READ state, \$00=Warmstart,
 \$80=Coldstart, \$40=APPLESOFT RAM

AA52 DOS CSWL intercept state number.

AA53 Address of true CSWL handler (2 bytes).
AA55 Address of true KSWL handler (2 bytes).
AA57 MAXFILES value.
AA59 Save area for S, X, Y, and A registers when DOS is entered (4 bytes).
AA5D Command line index value (offset into line).
AA5E MON flags: (C=\$40, I=\$20, O=\$10)
AA5F Index of last command times 2.
AA60 Range length for LOAD and BLOAD (2 bytes).
AA62 Index of pending command, if any.
AA63 Scratch variable (counter, message index, etc.)
AA64 Index of current keyword.
AA65 Keywords present on command line flags.
AA66-AA74 Keyword values parsed from command and defaulted.
AA66 Volume (2 bytes)
AA68 Drive (2 bytes)
AA6A Slot (2 bytes)
AA6C Length (2 bytes)
AA6E Record (2 bytes)
AA70 Byte (2 bytes)
AA72 Address (2 bytes)
AA74 MON value (one byte)
AA75-AA92 Primary file name buffer
AA93-AAB0 Secondary (RENAME) file name buffer
AAB1-AAC0 DOS main routines constants and variables.
AAB1 MAXFILES default (\$03).
AAB2 Control-D (\$84).
AAB3 EXEC file active flag (\$00=not active).
AAB4 EXEC file buffer address (2 bytes).
AAB6 Active BASIC flag: \$00=INTEGER, \$40=APPLESOFT ROM, \$80=APPLESOFT RAM
AAB7 RUN intercepted flag.
AAB8 "APPLESOFT" characters in ASCII (9 bytes)
AAC1-AAC8 File manager constants.
AAC1 Address of RWTS paramter list (B7E8).
AAC3 Address of VTOC sector buffer (B3BB).
AAC5 Address of directory sector buffer (B4BB).
AAC7 Address of last byte of DOS plus one. (C000)
AAC9-AAE4 File manager function routine entry point table.
This table contains a two byte function handler routine address for each of the 14 file manager opcodes in opcode order.
AAE5-AAF0 File manager read subcode handler entry point table.
This table contains a two byte function handler routine address for each of the 6 read subcodes.
AAF1-AAFC File manager write subcode handler entry point table.
This table contains a two byte function handler routine address for each of the 6 write subcodes.
AAFD-AB05 File manager external entry point (from \$3D6).
Is X register zero?
If so, allow new files by simulating an INIT command index.
Otherwise, require old file by simulating a LOAD command index.
Fall through to main file manager entry point.
AB06-AB1E File manager main entry.
Save S register at B39B.
Restore file manager workarea from file buffer (AE6A)
Make sure opcode does not exceed 13.

If it does, return with code=2 (invalid opcode).
 Use opcode as index into file manager function
 routine entry point table and go to proper handler
 via RTS.

AB1F-AB21 Return with return code=2 (bad opcode).

AB22-AB27 OPEN function handler.
 Call common open code (AB28).
 Exit file manager.

AB28-ABDB Common open routine.
 Initialize file manager workarea by resetting
 variables to their defaults (ABDC).
 Set sector length to 256.
 Insure record length is non-zero. If zero, use 1.
 Store record length in file manager workarea.
 Locate or allocate a directory entry for the file
 (B1C9).
 If file already exists, go to ABA6.
 Otherwise, save directory index for free entry.
 Using last command index and valid keywords table,
 determine whether current command may create a new
 file.
 If so, go to AB64.
 Otherwise, if running "APPLESOFT", set return code
 to "LANGUAGE NOT AVAILABLE" and exit.
 If not running "APPLESOFT" set return code to "FILE
 NOT FOUND" and exit.

AB64 Set sector count in directory entry to 1 (there will
 only be a T/S List sector initially).
 Allocate a sector for a T/S List (B244).
 Store sector number of this sector in directory
 entry and in first and current T/S List sector number
 in file manager workarea.
 Store track number in both places also.
 Move file type desired to directory entry.
 Write directory sector back to catalog (B037).
 Select T/S List buffer (AF0C).
 Zero it (B7D6).
 And write it back (AF3A).
 Set return code to 6 ("FILE NOT FOUND").

ABA6 Place track/sector of T/S List in directory entry in
 first T/S List variable in file manager workarea.
 Copy file type from directory to parmlist to pass it
 back to caller and to file manager workarea.
 Copy number of sectors in file to workarea.
 Save directory offset to entry in workarea.
 Set end of data pointer to "infinity".
 Set number of data bytes represented by one T/S List
 sector to 122*256 (30.5K) in workarea.
 Go read first T/S List sector (AF5E).

ABDC-AC05 Initialize file manager workarea.
 Zero entire 45 bytes of workarea.
 Save complemented volume number in workarea.
 Save drive number in workarea.
 Save slot*16 in workarea.
 Set track number to \$11 (catalog track).
 Return to caller.

AC06-AC39 CLOSE function handler.
 Checkpoint data buffer to disk if needed (AF1D).
 Checkpoint T/S List buffer if needed (AF34).

Release any sectors which were preallocated but not used (B2C3).
If VTOC does not need to be re-read, exit.
Otherwise, re-read VTOC sector (AFF7).
Flush through directory sectors in the catalog until we reach the one which contains the entry for this file.
Get the index to the entry.
Update the sector count in the entry to reflect the new file's length.
Checkpoint the directory sector back to the disk.
Exit file manager.

AC3A-AC57 RENAME function handler.
Call common code to locate/open the file.
If file is locked, exit with "FILE LOCKED" return code.
Set \$42,\$43 to point to new name.
Copy new name to directory entry.
Write back directory sector to disk.
Exit file manager.

AC58-AC69 READ function handler.
Insure subcode does not exceed 5. If so, exit with return code=3.
Use subcode as index into READ subcode handler entry point table.
Go to proper handler of subcode.

AC6A-AC6C Return code = 3, subcode bad

AC6D-AC6F "FILE LOCKED" error return

AC70-AC86 WRITE function handler.
If file is locked, exit with "FILE LOCKED" error.
Insure subcode does not exceed 5. If so, exit with return code=3.
Use subcode as index into WRITE subcode handler entry point table.
Go to proper handler of subcode.

AC87-AC89 POSITION AND READ ONE BYTE subcode handler
Call position routine.
Fall through to next subcode handler.

AC8A-AC92 READ ONE BYTE subcode handler.
Read next file byte (ACA8).
Store in parmlist for pass back to caller.
Exit the file manager.

AC93-AC95 POSITION AND READ A RANGE OF BYTES subcode handler.
Call position routine.
Fall through to next subcode handler.

AC96-ACA7 READ A RANGE OF BYTES subcode handler.
Decrement and check length (B1B5).
Read a byte (ACA8).
Point \$42,\$43 at range address and add one to address
Store byte read at address.
Loop back to AC96. (Length check will exit file manager when length is zero.)

ACA8-ACB8 Read a data byte.
Read next data sector if necessary (B0B6).
If at end of file, exit with "END OF DATA" error.
Otherwise, load data byte from data sector buffer.
Increment record number/byte offset into file (B15B).
Increment file position offset (B194).
Return with data byte read.

ACBB-ACBD POSITION AND WRITE ONE BYTE subcode handler.
 Call position routine.
 Fall through to next subcode handler.

ACBE-ACC6 WRITE ONE BYTE subcode handler.
 Find data byte to be written.
 Write it to file (ACDA).
 Exit file manager.

ACC7-ACC9 POSITION AND WRITE A RANGE OF BYTES subcode handler.
 Call position routine.
 Fall through to next subcode handler.

AACA-ACD7 WRITE A RANGE OF BYTES subcode handler.
 Copy and advance range address pointer.
 Get next byte to write.
 Write it to file (ACDA).
 Test and decrement length (B1B5).
 Loop back to AACA.

ACDA-ACEC Write a data byte.
 Read the proper data sector (if necessary) (B0B6).
 Store data byte to be written in sector buffer.
 Flag data sector buffer as requiring rewrite.
 Increment record number/byte offset into file (B15B).
 Exit via file position offset increment routine (B194).

ACEF-ACF5 LOCK function handler.
 Set mask byte to \$80 (lock).
 Go to common code (ACFB).

ACF6-ACFA UNLOCK function handler.
 Set mask byte to \$00 (unlock).
 Fall through to common code.

ACFB-AD11 LOCK/UNLOCK common code.
 Locate/open file (AB28).
 Get index into directory to entry.
 Update file type byte to lock (\$8X) or unlock (\$0X).
 Write directory sector back to disk.
 Exit file manager.

AD12-AD17 POSITION function handler.
 Call position routine.
 Exit file manager.

AD18-AD2A VERIFY function handler.
 Locate/open file (AB28).

AD1B Read next data sector.
 If at end of file, exit file manager.
 Otherwise, increment sector position.
 And loop back to AD1B.

AD2B-AD88 DELETE function handler.
 Locate/open file (AB28).
 Using directory index, determine if file is locked.
 If so, exit with "FILE LOCKED" error code.
 Copy T/S List sector's track number from directory to workarea and to last character of file name in the directory entry itself.
 Store a \$FF over T/S List sector's track number in directory entry to mark file deleted.
 Copy T/S List sector's sector number to workarea.
 Write directory sector back to disk.

AD54 Read next T/S List sector (AF5E).
 If no more exist, write VTOC and exit file manager.
 Otherwise, select T/S List buffer (AF0C).
 Index to first T/S pair.

AD5E If track number is zero or minus, skip it.
Otherwise, free the data sector by updating the VTOC
bit map (AD89).
Index to next T/S pair.
If more, go to AD5E.
Get T/S of next T/S List sector from this one.
Free this T/S List sector (AD89).
Go process next one, if any (go to AD54).
Otherwise, write VTOC and exit file manager.

AD89-AD97 Free a sector.
Call B2DD to deallocate sector in VTOC bit map.
Zero the sector allocation area of the workarea.
Return to caller.

AD98-AE2E CATALOG function handler.
Initialize file manager workarea (ABDC).
Set V value to zero (complimented=\$FF).
Read the VTOC sector (AFF7).
Set up a counter for 22 lines before waiting for
the keyboard.
Skip 2 lines on the screen.
Print "DISK VOLUME ".
Convert Volume number and print it (AE42).
Skip 2 more lines.

ADCA Read next directory sector.
If no more exist, exit file manager.
Set index to first entry.

ADD1 Get track number.
If zero, exit file manager.
If minus, skip entry (deleted file).
Print "*" if file is locked (check file type byte).
Use file type as index into file type name table at
B3A7 and print single character found there.
Print a blank.
Convert and print the number of sectors in the file.
Print a blank.
Index to filename.
Print file name.
Skip to next line.
Advance index to next directory entry.
If there are more, go to ADD1.
If not, go to ADCA to read next directory sector.
Exit when finished.

AE2F-AE41 Skip a line on CATALOG printout.
Output a carriage return.
Decrement line counter.
If still nonzero, exit.
Otherwise, wait for keyboard keypush.
Then reset counter to 21 lines.
And return to caller.

AE42-AE69 Convert the number stored at \$44 to a three character
printable number and print it.

AE6A-AE7D Restore file manager workarea from file buffer.
Select file manager workarea buffer.
Set return code in parmlist to zero initially.
Copy 45 byte saved image of file manager workarea in
file buffer to real file manager workarea.
Exit to caller.

AE7E-AE8D Save file manager workarea in file buffer.
Select file manager workarea buffer.

Copy 45 byte workarea to file buffer.
Exit to caller.

AE8E-AF07 INIT function handler.
Initialize the file manager workarea (ABDC).
Call RWTS to format the diskette (B058).
Copy V value to VTOC buffer.
Start track to allocate next value at \$11.
And direction of allocation as \$01 (forward).
Zero VTOC bit map (all sectors in use).
Skipping the first three tracks and track \$11, copy the 4 byte bit mask (B3A0) to each track entry in the VTOC bit map to free the sectors. This leaves the first three tracks and the catalog track marked in use.
Zero the directory sector buffer.
Point to directory sector buffer.
Set track \$11 in RWTS parmlist.
Set up link from this directory sector to next (track \$11, sector-1).
Call RWTS to write directory sector.
Write each sector on track in this way except for sector zero.
On last sector (sector 1) zero link pointer.
Point RWTS parms at DOS load point (B7C2).
Write DOS image onto tracks 0-2 (B74A).
Exit file manager.

AF08-AF1C Select a buffer by setting \$42,\$43 to point to it.
AF08 Select file manager workarea buffer in file buffer.
AF0C Select T/S List sector buffer in file buffer.
AF10 Select data sector buffer in file buffer.
Exit to caller when \$42,\$43 are set.

AF1D-AF33 Checkpoint write data sector buffer to disk.
Test flag to see if buffer was changed since last read/write.
If not, exit to caller.
Otherwise, set up RWTS pointer (AFE4).
Call RWTS to write sector.
Reset flag to indicate data sector no longer in need of a checkpoint.
Exit to caller.

AF34-AF4A Checkpoint write T/S List sector buffer to disk.
Test flag to see if buffer was changed since last read/write.
If not, exit to caller.
Otherwise, set up RWTS pointer (AF4B).
Call RWTS to write sector.
Reset flag to indicate T/S List sector no longer in need of checkpoint.
Exit to caller.

AF4B-AF5D Prepare for RWTS call with a T/S List sector.
Copy address of T/S List buffer to RWTS parmlist.
Get track/sector of sector.
Exit to caller.

AF5E-AFDB Read a T/S List sector to file buffer.
(CARRY flag is set at entry to indicate whether the first T/S List for the file is wanted (C=0) or the next (C=1).
Memorize carry flag entry code.
Checkpoint current T/S List sector if necessary.

Set up for RWTS (AF4B).
Select T/S List buffer (AF0C).
Is first or next wanted?
If first, go to AFB5 to continue.
Otherwise, get link to next T/S List from this one.
If link is non-zero, use it to find next one and go to AFB5.
Otherwise, we are out of T/S Lists for this file.
If we are reading file, exit with error code.
Otherwise, allocate a new sector (B244).
Point old T/S List sector to new one's track/sector.
Write old T/S List sector back to disk.
Zero the buffer to form new T/S List sector.
Compute and store the relative sector number of the first sector listed in this sector at +5,+6 into the buffer.
Set RWTS opcode to write new T/S List sector to disk.

AFB5 Set RWTS opcode to read old T/S List (unless we just allocated it above).
Set track and sector and call RWTS to read old list or write new list.
Compute relative sector number of last sector (plus one) in this list and store in workarea.
Exit to caller with normal return code.

AFDC-AFE3 Read a data sector.
Set up for RWTS (AFE4).
Set RWTS READ opcode and go to RWTS driver to do it.

AFE4-AFF6 Prepare for RWTS with data sector.
Copy address of data sector buffer to RWTS parmlist.
Get its track/sector.
And exit to caller.

AFF7-B010 Read/write the VTOC buffer.
AFF7 Read VTOC entry, go to AFFD.
AFFB Write VTOC entry, fall through.
AFFD Common code.
Copy VTOC sector buffer address to RWTS parmlist.
Get its track number and use sector \$00.
Exit through RWTS driver.

B011-B036 Read a directory sector.
(If CARRY flag is zero on entry, read first directory sector. If CARRY is one, read next.)
Memorize entry code.
Set buffer pointers (B045).
First or next?
If first, get track/sector of directory sector from VTOC at offset +1,+2.
Otherwise, get track/sector from directory sector at offset +1,+2. If track is zero, exit with error code (end of directory).
Call RWTS to read sector.
Exit with normal return code.

B037-B044 Write directory sector.
Set buffer pointers.
Find its track/sector in workarea.
Exit through RWTS to write it.

B045-B051 Prepare for RWTS for directory buffer.
Copy directory buffer address to RWTS parmlist.
Exit to caller.

B052-B0B3 Read/Write Track/Sector (RWTS) driver.

```

    Set track/sector in RWTS parmlist.
B058 Set command code (read, write, etc.).
    If writing, set flag (B5D5).
    Set volume number expected in parmlist.
    Set slot*16 in parmlist.
    Set drive in parmlist.
    Set sector size in parmlist.
    Set IOB type in parmlist ($01).
    Call RWTS, passing parmlist pointer.
    Copy true volume found to file manager parmlist.
    Reset volume expected field in RWTS parmlist.
    If an error did not occur, exit to caller.
    Otherwise, get return code.
    Translate vol mismatch to rc=7, write protected to
    rc=4 and all other errors to rc=8 (I/O error).
    Exit file manager now.
B0B6-B133 Read next data sector (if necessary).
    Is the current file position in the current data
    sector now in memory?
    If so, go to B12C.
    Otherwise, checkpoint data sector buffer.
    Is the current file position prior to or after this
    T/S List's domain?
    If not, go to B0F3.
    Otherwise, read each T/S List for the file, starting
    with the first, until the proper one is found.
    If it is never found, exit with error (ran off end of
    file reading).
B0F3 Data is in this T/S List sector.
    Compute the displacement to the proper entry in this
    T/S List sector.
    Select the T/S List buffer.
    Get the track of the data sector wanted.
    If non-zero, go to B114.
    Otherwise, if not writing, exit with error (no data
    to read there).
    If writing, allocate a new sector and store its
    track/sector location in the list at this point
    (B134).
    Go to B120.
B114 Read old data sector, using the track/sector found
    in the T/S List entry.
B120 Save number of sector last read in workarea.
B12C Select data buffer.
    Get byte offset and exit normally to caller.
B134-B15A Add a new data sector to file.
    Allocate a sector (B244).
    Put track/sector numbers in T/S List entry.
    Select data buffer and zero it.
    Set flags to indicate that the T/S List sector and
    the data sector buffer require checkpoints.
    Exit to caller.
B15B-B193 Increment record number and byte offset into file.
    Copy current record number and byte offset to file
    manager parameter list to pass back to caller.
    Increment byte offset in workarea.
    If byte offset equals record length, set byte offset
    back to zero and increment record number.
    Return to caller.

```

B194-B1A1 Increment file position offset.
Increment byte offset into current sector by one.
If at end of sector, increment sector number by one.
Return to caller.

B1A2-B1B4 Copy and advance range address.
Copy range address from file manager parmlist to \$42.
Increment range address in parmlist for next time through.
Return to caller.

B1B5-B1C8 Decrement range length.
Decrement range length in file manager parmlist by one.
If zero, exit file manager.
Otherwise, exit to caller.

B1C9-B21B Locate or allocate a directory entry in the catalog.
Read the VTOC sector (AFF7).
Set \$42,\$43 to point to file name we are looking for.
Set pass number to one (locate file).

B1D8 Initialize directory sector offset (first sector).

B1E1 Increment sector offset.
Read directory sector.
If at end of directory, go to B23A.
Set entry index to first file entry.

B1EB Get track.
If deleted, skip entry, go to B217.
If empty, end of directory, go to B212.
Advance index to filename in directory.
Compare against filename wanted.
If they match, return entry index and exit.
If not, advance index to next entry in sector and loop back to B1EB.
If at end of sector, go to B1E1 to get next sector.

B212 If pass number is one, go to B1D8 to start second pass.

B217 If pass number is one, go to B20B to skip entry.
If second pass, fall through to allocate entry.

B21C-B22F Copy file name to directory entry.
Advance index to file name field in directory entry.
Copy 30 byte filename to directory entry.
Reload directory index and return to caller.

B230-B239 Advance index to next directory entry in sector.
Add 35 (length of entry) to index.
Test for end of sector and return to caller.

B23A-B243 Switch to second pass in directory scan.
If on pass one, switch to pass 2 and go to B1D8.
If on pass two, exit file manager with "DISK FULL" error.

B244-B2C2 Allocate a disk sector.
Is there a track currently allocated to this file?
If not, go to B26A to find a track with free sectors.

B249 Otherwise, decrement sector number to get next possible free sector number.
If there are no more sectors on this track, go to B265 to find a new track.
Otherwise, rotate the track bit mask by one position and get the bit for this sector.
If the sector is in use, loop back to B249.
Otherwise, add one to file's sector count.
Pass back sector number (track number is at B5F1).

And return to caller.

B265 Indicate no track is being used at present.

B26A Reset allocation flag to allow at least one complete search of all tracks for some space.
Read VTOC sector.

B272 Get last track allocated from and add direction value to get next track to examine (+1 or -1).
Are we back to track 0?
If so, go to B284.
Otherwise, are we past track 34?
If so, reverse direction and go to B28E.

B284 Is this the second time we have come to track 0 ?
(check allocation flag).
If so, exit with "DISK FULL" error.
Otherwise, set allocation flag to remember this.
Set direction to forward (+1).

B28E Begin at directory track (17 + or - 1).
Compute bit map index (tracknumber*4).
Copy track bit map from VTOC to workarea, watching to see if all four bytes are zero (track is full).
In any case, set all four bytes in VTOC to zero (allocate all sectors).
If no free sectors in the track, go to B272 to try next track.
Otherwise, write VTOC to disk to insure file's integrity.
Set sector number to last sector in track.
Go to B249 to allocate one of its free sectors to the file.

B2C3-B2DC Release pre-allocated sectors in current track and checkpoint the VTOC.
Has a track been allocated to the file?
If not, exit to caller.
Otherwise, read VTOC.
Get next sector which could have been used (number of times track map was shifted during allocation).
Call B2DD to shift track bit map back and merge it back into the VTOC bit map.
Indicate no track has been allocated.
Exit to caller.

B2DD-B2FF Free one or more sectors by shifting mask in file manager's allocation area back into VTOC bit map.
(If CARRY is set, current sector is freed also)
Rotate entire 4 byte track bit mask once.
Repeat for as many sectors as were allocated.
Compute index into VTOC for this track's map.
If zero, exit.
Merge ("OR") file manager's bits with those already in VTOC, freeing sectors which were never used by the file.
Return to caller.

B300-B35E Calculate file position.
Set record number passed in file manager parmlist in workarea and in sector offsets.
Clear sector offset high part.
Perform a 16 bit multiply as follows:
3 byte file position = record number times record length.
Add the byte offset from the parmlist into the three

byte file position value (B5E4,B5E5,B5E6).
Return to caller.

B35F-B37D Error exits.

B35F RC=1 "LANGUAGE NOT AVAILABLE"
B363 RC=2 "RANGE ERROR" (bad opcode)
B367 RC=3 "RANGE ERROR" (bad subcode)
B36B RC=4 "WRITE PROTECTED"
B36F RC=5 "END OF DATA"
B373 RC=6 "FILE NOT FOUND"
B377 RC=9 "DISK FULL" (all files closed)
B37B RC=A "FILE LOCKED"

B37F-B396 Exit file manager.

B37F Exit with no errors.
Get return code of zero.
Clear carry flag and go to B386.
B385 Set carry flag to indicate error.
B386 Save return code in parmlist.
Clear monitor status register (\$48) after RWTS has probably tromped on it.
Save file manager workarea to file buffer (AE7E).
Restore processor status and stack register.
Exit to original caller of file manager.

B397-B3A3 File manager scratch space.

B397 Track/sector of current directory sector (2 bytes).
B39B S register save area.
B39C Directory index.
B39D Catalog line counter/Directory lookup flag/Etc.
B39E LOCK/UNLOCK mask/Allocation flag/Etc.
B3A0 Four byte mask used by INIT to free an entire track in the VTOC bit map.

B3A4-B3A6 Decimal conversion table (1,10,100).
B3A7-B3AE File type name table used by CATALOG.
File types are: T,I,A,B,S,R,A,B, corresponding to hex values: \$00, \$01, \$02, \$04, \$08, \$10, \$20, and \$40 respectively.

B3AF-B3BA ASCII text "DISK VOLUME " backwards. Used by CATALOG.

B3BB-B4BA VTOC sector buffer.

B3BC Track/sector of first directory sector.
B3BE DOS release number (1, 2, or 3).
B3C1 Volume number of diskette.
B3E2 Number of entries in each T/S List sector.
B3EB Track to allocate next.
B3EC Direction of track allocation (+1 or -1)
B3EF Number of tracks on a disk.
B3F0 Number of sectors on a disk.
B3F1 Sector size in bytes (2 bytes)
B3F3 Track 0 bit map
B3F7 Track 1 bit map
etc.
B47B Track 34 bit map

B4BB-B5BA DIRECTORY sector buffer.

B4BC Track/sector of next directory sector.
B4C6 First directory entry and
Track of T/S List
B4C7 Sector of T/S List
B4C8 File type and lock bit
B4C9 Filename field (30 bytes)
B4E7 Size of file in sectors (including T/S List(s)).

B5BB-B5D0 File manager parameter list.

B5BB Opcode
 B5BC Subcode
 B5BD Eight bytes of variable parameters depending on opcode.
 B5C5 Return code.
 B5C7 Address of file manager workarea buffer.
 B5C9 Address of T/S List sector buffer.
 B5CB Address of data sector buffer.
 B5CD Address of next DOS buffer on chain (not used).
 B5D1-B5FD File manager workarea.
 B5D1 1st T/S List sector's track/sector.
 B5D3 Current T/S List sector's track/sector.
 B5D5 Flags: 80=T/S List needs checkpoint
 40=Data sector needs checkpoint
 20=VTOC sector needs checkpoint
 02=Last operation was write
 B5D6 Current data sector's track/sector.
 B5D8 Directory sector index for file entry.
 B5D9 Index into directory sector to directory entry for file.
 B5DA Number of sectors described by one T/S List.
 B5DC Relative sector number of first sector in list.
 B5DE Relative sector number +1 of last sector in list.
 B5E0 Relative sector number of last sector read.
 B5E2 Sector length in bytes.
 B5E4 File position (3 bytes) sector offset, byte offset into that sector.
 B5E8 Record length from OPEN.
 B5EA Record number.
 B5EC Byte offset into record.
 B5EE Number of sectors in file.
 B5F0 Sector allocation area (6 bytes).
 Next sector to allocate (shift count)
 Track being allocated
 Four byte bit map of track being allocated, rotated to next sector to allocate.
 B5F6 File type.
 B5F7 Slot number times 16.
 B5F8 Drive number.
 B5F9 Volume number (complemented).
 B5FA Track number.
 B5FE-B5FF Not used.
 B600-B6FF Start of Boot 2/RWTS image.
 B600 Boot 1 image which can be written to INITED disks on track 0, sector 0.
 B65D DOS 3.3 patch area.
 B65D APPEND patch flag.
 B65E APPEND patch. Come here when file manager driver gets an error other than end of data.
 Locate and free the file buffer.
 Clear the APPEND flag.
 Get the error number and go print error (A6D2).
 B671 APPEND patch. Come here from APPEND command handler to increment record number if APPEND flag is set and to clear the flag. Exit through POSITION.
 B686 VERIFY patch. Come here from I/O a range of bytes routine to exit through VERIFY after SAVE or BSAVE.
 B692 APPEND patch. Come here from file manager driver if return code was END OF DATA.

Test the file position for zero.
If non-zero, set APPEND flag on and return to caller.
If zero (at start of file), copy record number and
byte offset to file manager parmlist and return a
zero data byte to caller.

B6FE Page address of first page in Boot 2.
B6FF Number of sectors (pages) in Boot 2.

B700-B749 DOS 2nd stage boot loader.
Set RWTS parmlist to read DOS from disk.
Call Read/Write group of pages (\$B793).
Create new stack.
Call SETVIC (\$FE93) and SETKBD (\$FE89).
Exit to DOS coldstart (\$9D84).

B74A-B78C Put DOS on tracks 0-2.
Set RWTS parmlist to write DOS to disk.
Call Read/Write group of pages (\$B793).
Exit to caller.

B78D-B792 Unused.

B793-B7B4 Read/Write a group of pages.
call RWTS through external entry point (\$B7B5).
Exit to caller.

B7B5-B7C1 Disable interrupts and call RWTS.

B7C2-B7D5 Set RWTS parameters for writing DOS.

B7D6-B7DE Zero current buffer.
Zero 256 bytes pointed to by \$42,\$43.
Exit to caller.

B7DF-B7E7 DOS 2nd stage boot loader parmlist.
B7DF Unused.
B7E0 Number of pages in 2nd DOS load.
B7E1 Number of sectors to read/write.
B7E2 Number of pages in 1st DOS load.
B7E3 INIT DOS page counter.
B7E4 Pointer to RWTS parmlist (2 bytes).
B7E6 Pointer to 1st stage boot location (2 bytes).

B7E8-B7F8 RWTS parmlist.
B7E8 Table type. Must be \$01.
B7E9 Slot number times 16.
B7EA Drive number (\$01 or \$02).
B7EB Volume number expected (0 matches any volume).
B7EC Track number (\$00 to \$22).
B7ED Sector number (\$00 to \$0F).
B7EE Pointer to Device Characteristics Table (2 bytes).
B7F0 Pointer to user data buffer for READ/WRITE (2 bytes).
B7F2 Unused.
B7F3 Byte count for partial sector (use \$00 for 256).
B7F4 Command code: 0=SEEK, 1=READ, 2=WRITE, 4=FORMAT.
B7F5 Error code:(valid if carry set) \$10=Write protect,
\$20=Volume mismatch, \$40=Drive error, \$80=Read error.
B7F6 Volume number found.
B7F7 Slot number found.
B7F8 Drive number found.

B7F9-B7FA Unused.

B7FB-B7FE Device Characteristics Table (DCT).
B7FB Device type (should be \$00).
B7FC Phases per track (should be \$01).
B7FD Motor on time count (2 bytes - should be \$EF, \$D8).

B7FF Unused.

B800-B829 PRENIBBLE routine.
Converts 256 (8 bit) bytes to 342 (6 bit) "nibbles"

of the form 00XXXXXX.
 Pointer to page to convert stored at \$3E,\$3F.
 Data stored at primary and secondary buffers.
 On entry: \$3E,\$3F contain pointer to user data.
 On exit: A-reg:unknown
 X-reg:\$FF
 Y-reg:\$FF
 Carry set
 Exit to caller.

B82A-B8B7 WRITE routine.
 Writes preinitialized data from primary and secondary buffers to disk.
 Calls Write a byte subroutine.
 Writes 5 bytes autosync, starting data marks (\$D5/\$AA/\$AD), 342 bytes data, one byte checksum, and closing data marks (\$DE/\$AA/\$EB).
 Uses Write Translate Table (\$ba29).
 On entry: X-reg:Slot number times 16
 On exit: Carry set if error
 If no error:
 A-reg:unknown
 X-reg:unchanged
 Y-reg:\$00
 Carry clear
 Uses \$26,\$27,\$678
 Exit to caller.

B8B8-B8C1 Write a byte subroutine.
 Timing critical code used to write bytes at 32 cycle intervals.
 Exit to caller.

B8C2-B8DB POSTNIBBLE routine.
 Converts 342 (6 bit) "nibbles" of the form 00XXXXXX to 256 (8 bit) bytes.
 Nibbles stored at primary and secondary buffers.
 Pointer to data page stored at \$3E,\$3F.
 On entry: X-reg:Slot number times 16
 \$36,\$37:pointer to user data
 \$26:byte count in secondary buffer (\$00)
 On exit: A-reg:unknown
 X-reg:unknown
 Y-reg:byte count in secondary buffer
 Carry set
 Exit to caller.

B8DC-B943 READ routine.
 Read a sector of data from disk and store it at primary and secondary buffers. (First uses secondary buffer high to low, then primary low to high)
 On entry: X-reg:Slot times 16
 Read mode (Q6L,Q7L)
 On exit: Carry set if error.
 If no error:
 A-reg:\$AA
 X-reg:unchanged
 Y-reg:\$00
 Carry clear
 Uses \$26
 Exit to caller.

B944-B99F RDADR routine.
 Read an Address Field.

Reads starting address marks (\$D5/\$AA/\$96), address information (volume/track/sector/checksum), and closing address marks (\$DE/\$AA).

On entry: X-reg:Slot number times 16

Read mode (Q6L,Q7L)

On exit: Carry set if error.

If no error:

A-reg:\$AA

X-reg:unchanged

Y-reg:\$000

Carry clear

\$2F: Volume number found

\$2E: Track number found

\$2D: Sector number found

\$2C: Checksum found

Uses \$26,\$27

Exit to caller.

B9A0-B9FC SEEKABS routine.

Move disk arm to desired track.

Calls arm move delay subroutine (\$B9FD).

On entry: X-reg:Slot number times 16

A-reg:Desired track (halftrack for single phase disk).

\$478:Current track.

On exit: A-reg:unknown

X-reg:unchanged

Y-reg:unknown

\$2A and \$478:Final track

\$27:Prior track (if seek needed)

Uses: \$26,\$27,\$2A,\$2B

Exit to caller.

B9FD-BA10 Arm move delay subroutine.

Delays a specified number of 100 Usec intervals.

On entry: A-reg:number of 100 Usec intervals.

\$46,\$47:Should contain motor on time count (\$EF,\$D8) from Device Characteristics Table

\$478:Current track.

On exit: A-reg:\$000

X-reg:\$000

Y-reg:unchanged

Carry set

Exit to caller.

BA11-BA28 Arm move delay table.

Contains values of 100 μ sec intervals used during Phase-on and Phase-off of stepper motor.

BA29-BA68 Write Translate Table.

Contains 6 bit "nibbles" used to convert 8 bit bytes. Values range from \$96 to \$FF.

Codes with more than one pair of adjacent zeros or with no adjacent ones are excluded.

BA69-BA95 Unused.

BA96-BAFF Read Translate Table.

Contains 8 bit bytes used to convert 6 bit "nibbles". Values range from \$96 to \$FF.

Codes with more than one pair of adjacent zeros or with no adjacent ones are excluded.

BB00-BBFF Primary Buffer.

BC00-BC55 Secondary Buffer.

BC56-BCC3 Write Address Field during initialization.

Calls Write double byte subroutine.
 Writes number of autosync bytes contained in Y-reg,
 starting address marks (\$D5/\$AA/\$96), address
 information (volume/track/sector/checksum), closing
 address marks (\$DE/\$AA/\$EB).
 On entry: X-reg:Slot number times 16
 Y-reg:number of autosync to write
 \$3E: \$AA
 \$3F: sector number
 \$41: volume number
 \$44: track number
 On exit: A-reg:unknown
 X-reg:unchanged
 Y-reg:\$00
 Carry set
 Exit to caller.

BCC4-BCDE Write double byte subroutine.
 Timing critical code that encodes address information
 into even and odd bits and writes it at 32 cycle
 intervals.
 Exit to caller.

BCDF-BCFF Unused.

BD00-BD18 Main entry to RWTS.
 Upon entry, store Y-reg and A-reg at \$48,\$49 as
 pointers to the IOB.
 Initialize maximum number of recals at 1 and seeks
 at 4.
 Check if the slot number has changed. If not,
 branch to SAMESLOT at \$BD34.

BD19-BD33 Update slot number in IOB and wait for old drive
 to turn off.

BD34-BD53 SAMESLOT
 Enter read mode and read with delays to see if disk
 is spinning.
 Save result of test and turn on motor just in case.

BD54-BD73 Move pointers in IOB to zero page for future use.
 Device Characteristics Table pointer at \$3C,\$3D
 and data buffer pointer at \$3E,\$3F.
 Set up \$47 (motor on time) with \$D8 from DCT.
 Check if the drive number has changed. If not,
 branch to \$BD74.
 If so, change test results to show drive off.

BD74-BD8F Select appropriate drive and save drive being used
 as high bit of \$35. 1=drive 1, 0=drive 2.
 Get test results. If drive was on, branch to \$BD90.
 Wait for capacitor to discharge using MSWAIT
 subroutine at \$BA00.

BD90-BDAA Get destination track and go to it using MYSEEK
 subroutine at \$BE5A.
 Check test result again and if drive was on,
 branch to TRYTRK at \$BDAB.
 Delay for motor to come up to speed.

BDAB-BDBB TRYTRK
 Get command code.
 If null, exit through ALLDONE at \$BE46, turning drive
 off and returning to caller.
 If =4, branch to FORMDSK at \$BE0D.
 Otherwise, move low bit into carry (set=read,
 clear=write) and save value on status reg.

If write operation, data is pre-nibbled via a call to PRENIB16 at \$B800.

BDBC-BDEC Initialize maximum retries at 48 and read an Address Field via RDADR16 at \$B944.
If read was good, branch to RDRIGHT at \$BDED.
If bad read, decrement retries, and, if still some left try again. Else, prepare to recalibrate.
Decrement recal count. If no more, then indicate drive error via DRVERR at \$BE04.
Otherwise, reinitialize reseeks at 4 and recalibrate arm. Move to desired track and try again.

BDED-BE03 RDRIGHT
Verify on correct track. If so branch to RTTRK at \$BE10.
If not, set correct track via SETTRK subroutine at \$BE95 and decrement reseek count.
If not zero then reseek track. If zero, then recal.

BE04-BE0A DRVERR
Clean up stack and status reg.
Load A-reg with \$40 (drive error)
Goto HNDLERR at \$BE48.

BE0B-BE0C Used to branch to ALLDONE at \$BE46.

BE0D-BF0F FORMDSK
Jump to DSKFORM at \$BEAF.

BE10-BE25 RTTRK
Check volume number found against volume number wanted.
If no volume was specified, then no error.
If specified volume doesn't match, load A-reg with \$20 (volume mismatch error) and exit via HNDLERR at \$BE48.

BE26-BE45 CRCTVOL
Check to see if sector is correct.
Use ILFAV table at \$BFB8 for software sector interleaving.
If wrong sector, try again by branching back to TRYADR at \$BDC1.
If sector correct, find out what operation to do.
If write, branch to WRIT at \$BE51.
Otherwise, read data via READ16 (\$B8DC).
If read is good, then postnibble data via POSTNB16 (\$B8C2) and return to caller with no error.

BE46-BE47 ALLDONE
Skip over set carry instruction in HNDLERR.

BE48-BE50 HNDLERR
Set carry.
Store A-reg in IOB as return code.
Turn off motor.
Return to caller.

BE51-BE59 WRITE
Write a sector using WRITE16 (\$B82A).
If the write was good, exit via ALLDONE (\$BE46).
If bad write, load A-reg with \$10 (write protect error) and exit via HNDLERR (\$BE48).

BE5A-BE8D MYSEEK
Provides necessary housekeeping before going to SEEKABS routine.
Determines number of phases per track and stores track information in appropriate slot dependent

location.

BE8E-BE94 XTOY routine.
Put slot in Y-reg by transferring X-reg divided by 16 into Y-reg.

BE95-BEAE Set track number.

BEAF-BF0C INIT command handler
Provides setup for initializing a disk.
Get the desired volume number from the IOB.
Zero both the primary and secondary buffers.
Recalibrate the disk arm to track 0.
Set the number of sync bytes to be written between sectors to \$28 (40.).
Call TRACK WRITE routine for the actual formatting.
Allow 48 retries during initialization.
Double check that the first sector found is zero after calling TRACK WRITE.
Increment the track number after successfully formatting a track.
Loop back until 35 tracks are done.

BF0D-BF61 TRACK WRITE routine.
Start with sector zero.
Precede it with 128 self-sync bytes.
Follow them with sectors 0 through 15 in sequence.
Set retry count for verifying the track at 48.
Fill the sector initialization map with positive numbers.
Loop through a delay period to bypass most of the initial self-sync bytes.
Read the first Address Field found.
If the read is good and sector zero was found, enter the VERIFY TRACK routine.
Decrement the sync count by 2 (until it reaches 16 at which time it is decremented by 1).
If sync count is greater than or equal to 5, exit via \$BF71.
If not, set carry and return to caller.

BF62-BF87 VERIFY TRACK routine.
This routine reads all 16 sectors from the track that was just formatted.
If an error occurs during the read of either the Address Field or the Data Field, the number of retries is decremented.
The routine continues reading until retries is zero.
Calls Sector Map routine (\$BF88).

BF88-BFA7 Sector Map routine.
This routine marks the sector initialization map as each sector is verified.
If an error occurs, the routine exits through \$BF6C, which decrements the number of retries and continues if that value is greater than zero.
Upon completion of track zero, the sync count is decremented by two if it is at least 16.

BFA8-BFB7 Sector Initialization Map used to mark sectors as they are initialized.
Contains a \$30 prior to initialization of a track.
Value changed to \$FF as each sector is completed.

BFB8-BFC7 Sector Translate Table
Sector interleaving done with software.

BFC8-BFD6 Patch area starts here.

Patch from \$B741 to zero language card during boot.
Call SETVID (\$FE93).
Unprotect Language Card (if present).
Store \$00 at \$E000.
Exit through SETKBD (\$FE89) and DOS coldstart.

BFD9-BFDB Unused.

BFDC-BFE5 Patch called from \$A0E2.
Set three additional defaults (Byte offset=0).
Return to caller.

BFE6-BFEC Patch called from \$A6D5.
Call \$A75B to reset state and set warmstart flag.
Mark RUN not interrupted.
Return to caller.

BFED-BFFF Patch called from \$B377.
Call \$AE7E to save file manager workarea.
Restore stack.
Close all open files (\$A316).
Save stack again.
Exit through \$B385 ("DISK FULL ERROR").



DOS ZERO PAGE USAGE

BYTE	USE
24	Cursor horizontal (DOS)
26,27	Sector read buffer address (ROM) Scratch space (RWTS)
28,29	BASL/BASH (DOS)
2A	Segment merge counter (ROM,BOOT) Scratch space (RWTS)
2B	BOOT slot*16 (ROM) Scratch space (RWTS)
2C	Checksum from sector header (RWTS)
2D	Sector number from sector header (RWTS)
2E	Track number from sector header (RWTS)
2F	Volume number from sector header (RWTS)
33	Prompt character (DOS)
35	Drive number in high bit (RWTS)
36,37	CSWL,CSWH (DOS)
38,39	KSWL,KSWH (DOS)
3C	Workbyte (ROM) Merge workbyte (BOOT) Device characteristics table address (RWTS)
3D	Sector number (ROM) Device characteristics table address (RWTS)
3E,3F	Address of ROM sector-read subroutine (BOOT) Buffer address (RWTS)
40,41	DOS image address (BOOT) File buffer address (DOS)
41	Format track counter (RWTS)
42,43	Buffer address (DOS)
44,45	Numeric operand (DOS)
46,47	Scratch space (RWTS)
48,49	IOB address (RWTS)
4A,4B	INTEGER BASIC LOMEM address (DOS) Format diskette workspace (RWTS)
4C,4D	INTEGER BASIC HIMEM address (DOS)
67,68	APPLESOFT BASIC PROGRAM START (DOS)
69,6A	APPLESOFT BASIC VARIABLES START (DOS)
6F,70	APPLESOFT BASIC STRING START (DOS)
73,74	APPLESOFT BASIC HIMEM address (DOS)
76	APPLESOFT BASIC line number high (DOS)
AF,B0	APPLESOFT BASIC PROGRAM END (DOS)
CA,CB	INTEGER BASIC PROGRAM START (DOS)
CC,CD	INTEGER BASIC VARIABLES END (DOS)
D6	APPLESOFT BASIC PROGRAM protection flag (DOS)
D8,D9	INTEGER BASIC line number (DOS) APPLESOFT BASIC ONERR (DOS)



APPLEVISION on endless repeat...

Appendix A

Example Programs

This section is intended to supply the reader with utility programs which can be used to examine and repair diskettes. These programs are provided in their source form to serve as examples of the programming necessary to interface practical programs to DOS. The reader who does not know assembly language may also benefit from these programs by entering them from the monitor in their binary form and saving them to disk for later use. It should be pointed out that the use of 16 sector diskettes is assumed, although most of the programs can be easily modified to work under any version of DOS. It is recommended that, until the reader is completely familiar with the operation of these programs, it would be well advised to use them only on an “expendable” diskette. None of the programs can physically damage a diskette, but they can, if used improperly, destroy the data on a diskette, requiring it to be re-INITIALIZED.

Five programs are provided:

DUMP TRACK DUMP UTILITY

This is an example of how to directly access the disk drive through its I/O select addresses. **DUMP** may be used to dump any given track in its raw, preformatted form, to memory for examination. This can be useful both to understand how disks are formatted and in diagnosing clobbered diskettes.

ZAP DISK UPDATE UTILITY

This program is the backbone of any attempt to patch a diskette directory back together. It is also useful in examining the structure of files stored on disk and in applying patches to files or DOS directly. **ZAP** allows its user to read, and optionally write, any sector on a diskette. As such, it serves as a good example of a program which calls Read/Write Track/Sector (RWTS).

INIT REFORMAT A SINGLE TRACK

This program will initialize a single track on a diskette. Any volume number (\$00-\$FF) may be specified. **INIT** is useful in restoring a track whose sectoring has been damaged without reinitializing the entire diskette. DOS 3.3 and 48K is assumed.

FTS FIND T/S LISTS UTILITY

FTS may be used when the directory for a diskette has been destroyed. It searches every sector on a diskette for what appear to be Track/Sector Lists, printing the track and sector location of each it finds. Knowing the locations of the T/S Lists can help the user patch together a new catalog using **ZAP**.

COPY CONVERT FILES

COPY is provided as an example of direct use of the DOS File Manager package from assembly language. The program will read an input **B**-type file and copy its contents to an output **T**-type file. Although it could be used, for example, to convert files used by the Programma PIE editor for use by the Apple Toolkit assembler, it is not included as a utility program but rather as an example of the programming necessary to access the File Manager.

STORING THE PROGRAMS ON DISKETTE

The enterprising programmer may wish to type the source code for each program into an assembler and assemble the programs onto disk. The Apple Toolkit assembler was used to produce the listings presented here, and interested programmers should consult the documentation for that assembler for more information on the pseudo-opcodes used. For the non-assembly language programmer, the binary object code of each program may be entered from the monitor using the following procedure.

The assembly language listings consist of columns of information as follows:

The address of some object code
The object code which should be stored there
The statement number
The statement itself

For example...

```
0800:20 DC 03 112 COPY JSR LOCFPL FIND PARMLIST
```

indicates that the binary code “20DC03” should be stored at 0800 and that this is statement 112. To enter a program in the monitor, the reader must type in each address and its corresponding object code. The following is an example of how to enter the DUMP program:

```
CALL -151 (Enter the monitor from BASIC)  

0800:20 E3 03  

0803:84 00  

0805:85 01  

0807:A5 02  

...etc...  

0879:85 3F  

087B:4C B3 FD  

BSAVE DUMP,A$800,L$7E (Save program to disk)
```

Note that if a line (such as line 4 in DUMP) has no object bytes associated with it, it may be ignored. When the program is to be run...

```
BLOAD DUMP (Load program)  

CALL -151 (Get into monitor)  

02:11 N 800G (Store track to dump, run program)
```

The BSAVE commands which must be used with the other programs are:

```
BSAVE ZAP,A$900,L$6C  

BSAVE INIT,A$800,L$89  

BSAVE FTS,A$900,L$DC  

BSAVE COPY,A$800,L$1EC
```

DUMP – TRACK DUMP UTILITY

The **DUMP** program will dump any track on a diskette in its raw, pre-nibbilized format, allowing the user to examine the sector address and data fields and the formatting of the track. This allows the curious reader to examine diskettes to better understand the concepts presented in the preceeding chapters. **DUMP** may also be used to examine most protected disks to see how they differ from normal ones and to diagnose diskettes with clobbered sector address or data fields with the intention of recovering from disk I/O errors. The **DUMP** program serves as an example of direct use of the Disk II hardware from assembly language, with little or no use of DOS.

To use **DUMP**, first store the number of the track you wish dumped at location **\$02**, then begin execution at **\$800**. **DUMP** will return to the monitor after displaying the first part of the track in hexadecimal on the screen. The entire track image is stored, starting at **\$1000**. For example:

```
CALL -151      (Get into the monitor from BASIC)  
BLOAD DUMP    (Load the DUMP program)
```

...Now insert the diskette to be dumped...

```
02:11 N 800G   (Store a 11 (track 17, the catalog  
                  track) in $02, N terminates the store  
                  command, go to location $800)
```

The output might look like this...

```
1000- D5 AA 96 AA AB AA BB AB   (Start of sector address)  
1008- AA AB BA DE AA E8 C0 FF  
1010- 9E FF FF FF FF FF D5 AA   (Start of sector data)  
1018- AD AE B2 9D AC AE 96 96   (Sector data)  
...etc...
```

Quite often, a sector with an I/O error will have only one bit which is in error, either in the address or data header or in the actual data itself. A particularly patient programmer can, using **DUMP** and perhaps a half hour of hand “nibbilizing” determine the location of the error and record the data on paper for later entry via **ZAP**. A thorough understanding of Chapter 3 is necessary to accomplish this feat.

```

1  *****
2  *
3  * DUMP:THIS PROGRAM WILL ALLOW ITS *
4  *   USER TO DUMP AN ENTIRE       *
5  *   TRACK IN ITS RAW FORM INTO  *
6  *   MEMORY FOR EXAMINATION.      *
7  *
8  * INPUT: $02 = TRACK TO BE READ  *
9  *
10 * OUTPUT:$1000 = ADDRESS OF TRACK *
11 *   IMAGE
12 *
13 * ENTRY POINT: $800
14 *
15 * PROGRAMMER: DON D WORTH 2/19/81 *
16 *
17 *****

19 * ZPAGE DEFINITIONS

21 PTR      EQU    $0      WORK POINTER
22 TRACK    EQU    $2      TRACK TO BE READ/WITTEN
23 A1L      EQU    $3C     MONITOR POINTER
24 A2L      EQU    $3E     MONITOR POINTER
25 PREG     EQU    $48     MONITOR STATUS REGISTER

27 * OTHER ADDRESSES

29 BUFFER   EQU    $1000    TRACK IMAGE AREA
30 LOCRLPL  EQU    $3E3     LOCATE RWTS PARMLIST SUBRTN
31 RWTS     EQU    $3D9     RWTS SUBROUTINE
32 COUT     EQU    $FDED    PRINT ONE CHAR SUBROUTINE
33 XAM      EQU    $FDB3    MONITOR HEX DUMP SUBRTN

35 * DISK I/O SELECTS

37 DRVSM0   EQU    $C080    STEP MOTOR POSITIONS
38 DRVSM1   EQU    $C081
39 DRVSM2   EQU    $C082
40 DRVSM3   EQU    $C083
41 DRVSM4   EQU    $C084
42 DRVSM5   EQU    $C085
43 DRVSM6   EQU    $C086
44 DRVSM7   EQU    $C087
45 DRVOFF   EQU    $C088    TURN DRIVE OFF AFTER 6 REVS
46 DRVON    EQU    $C089    TURN DRIVE ON
47 DRVSL1   EQU    $C08A    SELECT DRIVE 1
48 DRVSL2   EQU    $C08B    SELECT DRIVE 2
49 DRVVD    EQU    $C08C    READ DATA LATCH
50 DRVVR    EQU    $C08D    WRITE DATA LATCH
51 DRVVRDM  EQU    $C08E    SET READ MODE
52 DRVVRM   EQU    $C08F    SET WRITE MODE

54 * RWTS PARMLIST DEFINITION

56          ORG    $0
0000: 00    57 RPLIOB  DS    1      IOB TYPE ($01)
0001: 00    58 RPLSLT  DS    1      SLOT*16
0002: 00    59 RPLDRV  DS    1      DRIVE
0003: 00    60 RPLVOL  DS    1      VOLUME
0004: 00    61 RPLTRK  DS    1      TRACK
0005: 00    62 RPLSEC  DS    1      SECTOR
0006: 00 00    63 RPLDCT  DS    2      ADDRESS OF DCT
0008: 00 00    64 RPLBUF  DS    2      ADDRESS OF BUFFER
000A: 00 00    65 RPLSIZ  DS    2      SECTOR SIZE

```

000C: 00	66	RPLCMD	DS	1	COMMAND CODE
	67	RPLCNL	EQU	\$00	NULL COMMAND
	68	RPLCRD	EQU	\$01	READ COMMAND
	69	RPLCWR	EQU	\$02	WRITE COMMAND
	70	RPLCFM	EQU	\$04	FORMAT COMMAND
000D: 00	71	RPLRCD	DS	1	RETURN CODE
	72	RPLRWP	EQU	\$10	WRITE PROTECTED
	73	RPLRVM	EQU	\$20	VOLUME MISMATCH
	74	RPLRDE	EQU	\$40	DRIVE ERROR
	75	RPLRRE	EQU	\$80	READ ERROR
000E: 00	76	RPLTVL	DS	1	TRUE VOLUME
000F: 00	77	RPLPSL	DS	1	PREVIOUS SLOT
0010: 00	78	RPLPDR	DS	1	PREVIOUS DRIVE
	79		ORG	\$800	
	81	* USE RWTS TO POSITION THE ARM TO THE DESIRED TRACK			
0800: 20 E3 03	83	DUMP	JSR	LOCRLP	LOCATE RWTS PARMLIST
0803: 84 00	84		STY	PTR	AND SAVE POINTER
0805: 85 01	85		STA	PTR+1	
0807: A5 02	87		LDA	TRACK	GET TRACK TO READ/WRITE
0809: A0 04	88		LDY	#RPLTRK	STORE IN RWTS LIST
080B: 91 00	89		STA	(PTR),Y	
080D: A9 00	91		LDA	#RPLCNL	NULL OPERATION
080F: A0 0C	92		LDY	#RPLCMD	AND STORE IN LIST
0811: 91 00	93		STA	(PTR),Y	
0813: A9 00	95		LDA	#0	ANY VOLUME WILL DO
0815: A0 03	96		LDY	#RPLVOL	
0817: 91 00	97		STA	(PTR),Y	
0819: 20 E3 03	98		JSR	LOCRLP	RELOAD POINTER TO PARMS
081C: 20 D9 03	99		JSR	RWTS	CALL RWTS
081F: A9 00	100		LDA	#0	
0821: 85 48	101		STA	PREG	FIX P REG SO DOS IS HAPPY
	103	* PREPARE TO DUMP TRACK TO MEMORY			
0823: A0 01	105		LDY	#RPLSLT	GET SLOT*16
0825: B1 00	106		LDA	(PTR),Y	
0827: AA	107		TAX		
0828: BD 89 C0	108		LDA	DRVON,X	KEEP DRIVE ON
082B: BD 8E C0	109		LDA	DRVDRM,X	INSURE READ MODE
082E: A9 00	111		LDA	#<BUFFER	POINT AT DATA
0830: 85 00	112		STA	PTR	
0832: A9 10	113		LDA	#>BUFFER	
0834: 85 01	114		STA	PTR+1	
0836: A0 00	115		LDY	#0	
	117	* START DUMPING AT THE BEGINNING OF A SECTOR ADDRESS			
	118	* FIELD OR A SECTOR DATA FIELD			
0838: BD 8C C0	120	LOOP1	LDA	DRVDR,X	WAIT FOR NEXT BYTE
083B: 10 FB	121		BPL	LOOP1	
083D: C9 FF	122		CMP	#\$FF	AUTOSYNC?
083F: D0 F7	123		BNE	LOOP1	NO, DON'T START IN MIDDLE
0841: BD 8C C0	124	LOOP2	LDA	DRVDR,X	WAIT FOR NEXT BYTE
0844: 10 FB	125		BPL	LOOP2	
0846: C9 FF	126		CMP	#\$FF	TWO AUTOSYNCS?
0848: D0 EE	127		BNE	LOOP1	NOT YET
084A: BD 8C C0	128	LOOP3	LDA	DRVDR,X	
084D: 10 FB	129		BPL	LOOP3	
084F: C9 FF	130		CMP	#\$FF	STILL AUTOSYNCS?

```

0851: F0 F7      131      BEQ    LOOP3      YES, WAIT FOR DATA BYTE
0853: D0 05      132      BNE    LOOP4      ELSE, START STORING DATA

                                134 * ONCE ALIGNED, BEGIN COPYING THE TRACK TO MEMORY.
                                135 * COPY AT LEAST TWICE ITS LENGTH TO INSURE WE GET IT
                                136 * ALL.

0855: BD 8C C0    138  LOOPD    LDA    DRVRD,X      WAIT FOR NEXT DATA BYTE
0858: 10 FB      139          BPL    LOOPD
085A: 91 00      140  LOOP4    STA    (PTR),Y      STORE IN MEMORY
085C: E6 00      141          INC    PTR          BUMP POINTER
085E: D0 F5      142          BNE    LOOPD
0860: E6 01      143          INC    PTR+1
0862: A5 01      144          LDA    PTR+1
0864: C9 40      145          CMP    #$40          DONE AT LEAST A TRACK?
0866: 90 ED      146          BCC    LOOPD        NO, CONTINUE
0868: BD 88 C0    147          LDA    DRVOFF,X      TURN DRIVE OFF

                                149 * WHEN FINISHED, DUMP SOME OF TRACK IN HEX ON SCREEN

086B: A9 00      151  EXIT     LDA    #<BUFFER      DUMP 800.8AF
086D: 85 3C      152          STA    A1L
086F: A9 10      153          LDA    #>BUFFER
0871: 85 3D      154          STA    A1L+1
0873: A9 AF      155          LDA    #<BUFFER+$AF
0875: 85 3E      156          STA    A2L
0877: A9 10      157          LDA    #>BUFFER+$AF
0879: 85 3F      158          STA    A2L+1
087B: 4C B3 FD    159          JMP    XAM          EXIT VIA HEX DISPLAY

```

--End assembly, 143 bytes, Errors: 0

ZAP – DISK UPDATE UTILITY

The next step up the ladder from **DUMP** is to access data on the diskette at the sector level. The **ZAP** program allows its user to specify a track and sector to be read into memory. The programmer can then make changes in the image of the sector in memory and subsequently use **ZAP** to write the modified image back over the sector on disk. **ZAP** is particularly useful when it is necessary to patch up a damaged directory. Its use in this regard will be covered in more detail when **FTS** is explained.

To use **ZAP**, store the number of the track and sector you wish to access in **\$02** and **\$03** respectively. Tracks may range from **\$00** to **\$22** and sectors from **\$00** to **\$0F**. For example, the Volume Table of Contents (VTOC) for the diskette may be examined by entering **\$11** for the track and **\$00** for the sector. **\$04** should be initialized with either a **\$01** to indicate that the sector is to be read into memory, or **\$02** to ask that memory be written out to the sector. Other values for location **\$04** can produce damaging results (**\$04** in location **\$04** will **INIT** your diskette!). When these three memory locations have been set up, begin execution at **\$900**. **ZAP** will read or write the sector into or from the 256 bytes starting at **\$800**. For example:

```
CALL -151          (Get into the monitor from BASIC)
BLOAD ZAP          (Load the ZAP program)
```

...Now insert the diskette to be zapped...

```
02:11 00 01 N 900G (Store a 11 (track 17, the catalog
                  track) in $02, a 00 (sector 0) at $03,
                  and a 01 (read) at $04. N ends the
                  store command and 900G runs ZAP.)
```

The output might look like this...

```
0800- 04 11 0F 03 00 00 01 00    (Start of VTOC)
0808- 00 00 00 00 00 00 00 00
0810- 00 00 00 00 00 00 00 00
0818- 00 00 00 00 00 00 00 00
...etc...
```

In the above example, if the byte at offset 3 (the version of DOS which **INIT**ed this diskette) is to be changed, the following would be entered...

```
003:02          (Change 03 to 02)
04:02 N 900G    (Change ZAP to write mode and do it)
```

Note that **ZAP** will remember the previous values in **\$02**, **\$03**, and **\$04**.

If something is wrong with the sector to be read (an I/O error, perhaps), ZAP will print an error message of the form:

RC=10

A return code of 10, in this case, means that the diskette was WRITE PROTECTED and a write operation was attempted. Other error codes are 20 (VOLUME MISMATCH), 40 (DRIVE ERROR), and 80 (READ ERROR). Refer to the documentation on RWTS given in Chapter 6 for more information on these errors.

```

1  *****
2  *
3  * ZAP: THIS PROGRAM WILL ALLOW ITS *
4  *   USER TO READ/WRITE           *
5  *   INDIVIDUAL SECTORS FROM/TO    *
6  *   THE DISKETTE                  *
7  *
8  * INPUT: $02 = TRACK TO BE READ   *
9  *           $03 = SECTOR TO BE READ *
10 *           /WRITTEN                *
11 *           $04 = $01 - READ SECTOR *
12 *                $02 - WRITE SECTOR *
13 *           $800 = ADDRESS OF SECTOR *
14 *                DATA BUFFER       *
15 *
16 * ENTRY POINT: $900
17 *
18 * PROGRAMMER: DON D WORTH 2/15/81 *
19 *
20 *****

22 BELL      EQU    $87      BELL CHARACTER

24 *          ZPAGE DEFINITIONS

26 PTR       EQU    $0       WORK POINTER
27 TRACK     EQU    $2       TRACK TO BE READ/WRITTEN
28 SECTOR    EQU    $3       SECTOR TO BE READ/WRITTEN
29 OPER      EQU    $4       OPERATION TO BE PERFORMED
30 READ      EQU    1        READ OPERATION
31 WRITE     EQU    2        WRITE OPERATION
32 A1L       EQU    $3C      MONITOR POINTER
33 A2L       EQU    $3E      MONITOR POINTER
34 PREG      EQU    $48      MONITOR STATUS REGISTER

36 *          OTHER ADDRESSES

38 BUFFER    EQU    $800     SECTOR DATA BUFFER
39 LOCRPL    EQU    $3E3     LOCATE RWTS PARMLIST SUBRTN
40 RWTS      EQU    $3D9     RWTS SUBROUTINE
41 COUT      EQU    $FDED     PRINT ONE CHAR SUBROUTINE
42 PRBYTE    EQU    $FDDA     PRINT ONE HEX BYTE SUBRTN
43 XAM       EQU    $FDB3     MONITOR HEX DUMP SUBRTN

45 *          RWTS PARMLIST DEFINITION

47          ORG    $0
0000: 00    48 RPLIOB DS    1      IOB TYPE ($01)
0001: 00    49 RPLSLT DS    1      SLOT*16
0002: 00    50 RPLDRV DS    1      DRIVE

```

0003: 00	51	RPLVOL	DS	1	VOLUME
0004: 00	52	RPLTRK	DS	1	TRACK
0005: 00	53	RPLSEC	DS	1	SECTOR
0006: 00 00	54	RPLDCT	DS	2	ADDRESS OF DCT
0008: 00 00	55	RPLBUF	DS	2	ADDRESS OF BUFFER
000A: 00 00	56	RPLSIZ	DS	2	SECTOR SIZE
000C: 00	57	RPLCMD	DS	1	COMMAND CODE
	58	RPLCNL	EQU	\$00	NULL COMMAND
	59	RPLCRD	EQU	\$01	READ COMMAND
	60	RPLCWR	EQU	\$02	WRITE COMMAND
	61	RPLCFM	EQU	\$04	FORMAT COMMAND
000D: 00	62	RPLRCD	DS	1	RETURN CODE
	63	RPLRWP	EQU	\$10	WRITE PROTECTED
	64	RPLRVM	EQU	\$20	VOLUME MISMATCH
	65	RPLRDE	EQU	\$40	DRIVE ERROR
	66	RPLRRE	EQU	\$80	READ ERROR
000E: 00	67	RPLTVL	DS	1	TRUE VOLUME
000F: 00	68	RPLPSL	DS	1	PREVIOUS SLOT
0010: 00	69	RPLPDR	DS	1	PREVIOUS DRIVE
	70	ORG		\$900	
	72	*	FILL IN RWTS LIST		
0900: 20 E3 03	74	ZAP	JSR	LOCRPL	LOCATE RWTS PARMLIST
0903: 84 00	75		STY	PTR	AND SAVE POINTER
0905: 85 01	76		STA	PTR+1	
0907: A5 02	78		LDA	TRACK	GET TRACK TO READ/WRITE
0909: A0 04	79		LDY	#RPLTRK	STORE IN RWTS LIST
090B: 91 00	80		STA	(PTR),Y	
090D: A5 03	82		LDA	SECTOR	GET SECTOR TO READ/WRITE
090F: C9 10	83		CMP	#16	BIGGER THAN 16 SECTORS?
0911: 90 04	84		BCC	SOK	NO
0913: A9 00	85		LDA	#0	
0915: 85 03	86		STA	SECTOR	YES, PUT IT BACK TO ZERO
0917: A0 05	87	SOK	LDY	#RPLSEC	
0919: 91 00	88		STA	(PTR),Y	STORE IN RWTS LIST
091B: A0 08	90		LDY	#RPLBUF	
091D: A9 00	91		LDA	#<BUFFER	STORE BUFFER PTR IN LIST
091F: 91 00	92		STA	(PTR),Y	
0921: C8	93		INY		
0922: A9 08	94		LDA	#>BUFFER	
0924: 91 00	95		STA	(PTR),Y	
0926: A5 04	97		LDA	OPER	GET COMMAND CODE
0928: A0 0C	98		LDY	#RPLCMD	AND STORE IN LIST
092A: 91 00	99		STA	(PTR),Y	
092C: A9 00	101		LDA	#0	ANY VOLUME WILL DO
092E: A0 03	102		LDY	#RPLVOL	
0930: 91 00	103		STA	(PTR),Y	
	105	*	NOW CALL RWTS TO READ/WRITE THE SECTOR		
0932: 20 E3 03	107		JSR	LOCRPL	RELOAD POINTER TO PARMS
0935: 20 D9 03	108		JSR	RWTS	CALL RWTS
0938: A9 00	109		LDA	#0	
093A: 85 48	110		STA	PREG	FIX P REG SO DOS IS HAPPY
093C: 90 1B	111		BCC	EXIT	ALL IS WELL
	113	*	ERROR OCCURED, PRINT "RC=XX"		
093E: A9 87	115		LDA	#BELL	BEEP THE SPEAKER
0940: 20 ED FD	116		JSR	COUT	

```

0943: A9 52      117      LDA    #'R'      PRINT THE "RC="
0945: 20 ED FD  118      JSR    COUT
0948: A9 43      119      LDA    #'C'
094A: 20 ED FD  120      JSR    COUT
094D: A9 3D      121      LDA    #'='
094F: 20 ED FD  122      JSR    COUT
0952: A0 0D      123      LDY    #RPLRCD
0954: B1 00      124      LDA    (PTR),Y    GET RWTS RETURN CODE
0956: 20 DA FD  125      JSR    PRBYTE    PRINT RETURN CODE IN HEX

```

```

127 *      WHEN FINISHED, DUMP SOME OF SECTOR IN HEX

```

```

0959: A9 00      129  EXIT    LDA    #<BUFFER    DUMP 800.8B7
095B: 85 3C      130      STA    A1L
095D: A9 08      131      LDA    #>BUFFER
095F: 85 3D      132      STA    A1L+1
0961: A9 AF      133      LDA    #<BUFFER+$AF
0963: 85 3E      134      STA    A2L
0965: A9 08      135      LDA    #>BUFFER+$AF
0967: 85 3F      136      STA    A2L+1
0969: 4C B3 FD  137      JMP     XAM      EXIT VIA HEX DISPLAY

```

```

--End assembly, 125 bytes, Errors: 0

```

INIT – REFORMAT A SINGLE TRACK

Occasionally the sectoring information on a diskette can become damaged so that one or more sectors can no longer be found by DOS. To correct this problem requires that the sector address and data fields be re-formatted for the entire track thus affected. **INIT** can be used to selectively reformat a single track, thus avoiding a total re-**INIT** of the diskette. Before using **INIT**, the user should first attempt to write on the suspect sector (using **ZAP**). If **RWTS** refuses to write to the sector (**RC=40**), then **INIT** must be run on the entire track. To avoid losing data, all other sectors on the track should be read and copied to another diskette prior to reformatting. After **INIT** is run they can be copied back to the repaired diskette and data can be written to the previously damaged sector.

To run **INIT**, first store the number of the track you wish reformatted at location **\$02**, the volume number of the disk at location **\$03**, and then begin execution at **\$800**. **INIT** will return to the monitor upon completion. If the track can not be formatted for some reason (eg. physical damage or problems with the disk drive itself) a return code is printed. For example:

```
CALL -151      (Get into the monitor from BASIC)
BLOAD INIT     (Load the INIT program)
```

...Now insert the disk to be **INIT**-ed...

```
02:11 FE N 800G      (Store a 11 (track 17, the catalog
                      track) in $02, a volume number of
                      $FE (254) in $03, N terminates the
                      store command, go to location $800)
```

WARNING: DOS 3.3 must be loaded in the machine before running **INIT** and a 48K Apple is assumed. **INIT** will not work with other versions of DOS or other memory sizes.

```
1 *****
2 *
3 * INIT: THIS PROGRAM WILL ALLOW ITS *
4 * USER TO INITIALIZE A *
5 * SINGLE TRACK WITH ANY VOLUME *
6 * NUMBER DESIRED. *
7 *
8 * INPUT: $02 = TRACK TO BE INITIALIZED *
9 * $03 = VOLUME NUMBER *
10 *
11 * ENTRY POINT: $800 *
12 *
13 * PROGRAMMER: PIETER LECHNER 2/19/81 *
14 *
15 *****

17 * ZPAGE DEFINITIONS
```

```

19 PTR EQU $0 WORK POINTER
20 TRACK EQU $2 TRACK TO BE READ/Written
21 VOLUME EQU $3 VOLUME NUMBER
22 SECFND EQU $2D SECTOR FOUND BY RDADR16
23 AA EQU $3E ZPAGE CONSTANT FOR TIMING
24 VOL EQU $41 VOLUME USED BY WRADR16
25 TRK EQU $44 TRACK USED BY WRADR16
26 SYNCNT EQU $45 SYNC COUNT USED BY DSKF2
27 PREG EQU $48 MONITOR P REGISTER SAVEAREA
28 BELL EQU $87 ASCII BELL

30 * OTHER ADDRESSES

32 LOCRPL EQU $3E3 LOCATE RWTS PARMLIST SUBRTN
33 RWTS EQU $3D9 RWTS SUBROUTINE
34 RTRYCNT EQU $578 RETRY COUNT FOR DSKF2
35 NBUF1 EQU $BB00 PRIMARY SECTOR BUFFER
36 NBUF2 EQU $BC00 SECONDARY SECTOR BUFFER
37 READ16 EQU $B8DC READ DATA FIELD ROUTINE
38 RDADR16 EQU $B944 READ ADDRESS FIELD ROUTINE
39 DSKF2 EQU $BF0D FORMAT ONE TRACK ROUTINE
40 COUT EQU $FDED MONITOR CHARACTER OUTPUT
41 PRBYTE EQU $FDDA MONITOR HEX OUTPUT

43 * DISK I/O SELECTS

45 DRVSM0 EQU $C080 STEP MOTOR POSITIONS
46 DRVSM1 EQU $C081
47 DRVSM2 EQU $C082
48 DRVSM3 EQU $C083
49 DRVSM4 EQU $C084
50 DRVSM5 EQU $C085
51 DRVSM6 EQU $C086
52 DRVSM7 EQU $C087
53 DRVOFF EQU $C088 TURN DRIVE OFF AFTER 6 REVS
54 DRVON EQU $C089 TURN DRIVE ON
55 DRVSL1 EQU $C08A SELECT DRIVE 1
56 DRVSL2 EQU $C08B SELECT DRIVE 2
57 DRVRD EQU $C08C READ DATA LATCH
58 DRVWR EQU $C08D WRITE DATA LATCH
59 DRVRDM EQU $C08E SET READ MODE
60 DRVWRM EQU $C08F SET WRITE MODE

62 * RWTS PARMLIST DEFINITION

0000: 00 64 RPLIOB DS 1 IOB TYPE ($01)
0001: 00 65 RPLSLT DS 1 SLOT*16
0002: 00 66 RPLDRV DS 1 DRIVE
0003: 00 67 RPLVOL DS 1 VOLUME
0004: 00 68 RPLTRK DS 1 TRACK
0005: 00 69 RPLSEC DS 1 SECTOR
0006: 00 00 70 RPLDCT DS 2 ADDRESS OF DCT
0008: 00 00 71 RPLBUF DS 2 ADDRESS OF BUFFER
000A: 00 00 72 RPLSIZ DS 2 SECTOR SIZE
000C: 00 73 RPLCMD DS 1 COMMAND CODE
74 RPLCNL EQU $00 NULL COMMAND
75 RPLCRD EQU $01 READ COMMAND
76 RPLCWR EQU $02 WRITE COMMAND
77 RPLCFM EQU $04 FORMAT COMMAND
000D: 00 78 RPLRCD DS 1 RETURN CODE
79 RPLRWP EQU $10 WRITE PROTECTED
80 RPLRVM EQU $20 VOLUME MISMATCH
81 RPLRDE EQU $40 DRIVE ERROR
82

```

000E: 00	83	RPLRRE	EQU	\$80	READ ERROR
000F: 00	84	RPLTVL	DS	1	TRUE VOLUME
0010: 00	85	RPLPSL	DS	1	PREVIOUS SLOT
	86	RPLPDR	DS	1	PREVIOUS DRIVE
	87		ORG	\$800	
	89	*	USE RWTS TO POSITION THE ARM TO THE DESIRED TRACK		
0800: 20 E3 03	91	DUMP	JSR	LOCRLP	LOCATE RWTS PARMLIST
0803: 84 00	92		STY	PTR	AND SAVE POINTER
0805: 85 01	93		STA	PTR+1	
0807: A5 02	95		LDA	TRACK	GET TRACK TO READ/WRITE
0809: A0 04	96		LDY	#RPLTRK	STORE IN RWTS LIST
080B: 91 00	97		STA	(PTR),Y	
080D: A9 00	99		LDA	#RPLCNL	NULL OPERATION
080F: A0 0C	100		LDY	#RPLCMD	AND STORE IN LIST
0811: 91 00	101		STA	(PTR),Y	
0813: A9 00	103		LDA	#0	ANY VOLUME WILL DO
0815: A0 03	104		LDY	#RPLVOL	
0817: 91 00	105		STA	(PTR),Y	
0819: 20 E3 03	106		JSR	LOCRLP	RELOAD POINTER TO PARMS
081C: 20 D9 03	107		JSR	RWTS	CALL RWTS
081F: BD 89 C0	108		LDA	DRVON,X	LEAVE DRIVE ON
	110	*	ESTABLISH ENVIRONMENT FOR DSKF2 ROUTINE		
0822: A5 02	112		LDA	TRACK	PASS TRACK TO DSKF2
0824: 85 44	113		STA	TRK	
0826: A5 03	114		LDA	VOLUME	AND VOLUME
0828: 85 41	115		STA	VOL	
082A: A9 AA	116		LDA	#\$AA	STORE CONSTANT FOR ZPAGE..
082C: 85 3E	117		STA	AA	TIMING
082E: A9 28	118		LDA	#\$28	START WITH 40 SYNCs..
0830: 85 45	119		STA	SYN CNT	BETWEEN SECTORS
0832: A0 56	120		LDY	#\$56	
0834: A9 00	121		LDA	#\$00	
0836: 99 FF BB	122	ZNBUF2	STA	NBUF2-1,Y	ZERO SECONDARY BUFFER
0839: 88	123		DEY		
083A: D0 FA	124		BNE	ZNBUF2	
083C: 99 00 BB	125	ZNBUF1	STA	NBUF1,Y	AND PRIMARY BUFFER
083F: 88	126		DEY		
0840: D0 FA	127		BNE	ZNBUF1	
	129	*	INITIALIZE TRACK		
0842: 20 0D BF	131		JSR	DSKF2	FORMAT TRACK AND VERIFY
0845: A9 08	132		LDA	#\$08	IN CASE OF ERROR...
0847: B0 19	133		BCS	HNDERR	ERROR?
	135	*	READ SECTOR ZERO TO VERIFY FORMATTING		
0849: A9 30	137		LDA	#\$30	NO, DOUBLE CHECK TRACK
084B: 8D 78 05	138		STA	RTRYCNT	ALLOW 48 RETRIES
084E: 38	139	NOGOOD	SEC		
084F: CE 78 05	140		DEC	RTRYCNT	COUNT RETRIES
0852: F0 0E	141		BEQ	HNDERR	
0854: 20 44 B9	142		JSR	RDADR16	READ AN ADDRESS FIELD
0857: B0 F5	143		BCS	NOGOOD	ERROR, TRY AGAIN
0859: A5 2D	144		LDA	SECFND	IS THIS SECTOR ZERO?
085B: D0 F1	145		BNE	NOGOOD	NO, TRY AGAIN
085D: 20 DC B8	146		JSR	READ16	YES, READ DATA FIELD
0860: 90 1F	147		BCC	DONETRK	ALL IS WELL, DONE.

```

0862: A0 0D      148  HNDERR  LDY  #RPLRCD  ELSE, PHONEY UP A RC
0864: 91 00      149          STA  (PTR),Y

                                151  *      ERROR OCCURED, PRINT "RC=XX"

0866: A9 87      153          LDA  #BELL      BEEP THE SPEAKER
0868: 20 ED FD   154          JSR  COUT
086B: A9 52      155          LDA  #'R'      PRINT THE "RC="
086D: 20 ED FD   156          JSR  COUT
0870: A9 43      157          LDA  #'C'
0872: 20 ED FD   158          JSR  COUT
0875: A9 3D      159          LDA  #'='
0877: 20 ED FD   160          JSR  COUT
087A: A0 0D      161          LDY  #RPLRCD
087C: B1 00      162          LDA  (PTR),Y    GET RWTS RETURN CODE
087E: 20 DA FD   163          JSR  PRBYTE    PRINT RETURN CODE IN HEX

                                165  *      WHEN DONE, EXIT TO CALLER

0881: BD 88 C0   167  DONETRK LDA  DRVOFF,X  TURN DRIVE OFF
0884: A9 00      168          LDA  #$00
0886: 85 48      169          STA  PREG      CLEAR P REGISTER FOR DOS
0888: 60         170          RTS          ; RETURN TO CALLER

```

--End assembly, 154 bytes, Errors: 0

FTS – FIND T/S LISTS UTILITY

From time to time one of your diskettes will develop an I/O error smack in the middle of the catalog track. When this occurs, any attempt to use the diskette will result in an **I/O ERROR** message from DOS. Generally, when this happens, the data stored in the files on the diskette is still intact; only the pointers to the files are gone. If the data absolutely must be recovered, a knowledgeable Apple user can reconstruct the catalog from scratch. Doing this involves first finding the T/S Lists for each file, and then using **ZAP** to patch a catalog entry into track 16 for each file which was found. **FTS** is a utility which will scan a diskette for T/S Lists. Although it may flag some sectors which are not T/S Lists as being such, it will never miss a valid T/S List. Therefore, after running **FTS** the programmer must use **ZAP** to examine each track/sector printed by **FTS** to see if it is really a T/S List. Additionally, **FTS** will find every T/S List image on the diskette, even some which were for files which have since been deleted. Since it is difficult to determine which files are valid and which are old deleted files, it is usually necessary to restore all the files and copy them to another diskette, and later delete the duplicate or unwanted ones.

To run **FTS**, simply load the program and start execution at **\$900**. **FTS** will print the track and sector number of each sector it finds which bears a resemblance to a T/S List. For example:

```
CALL -151      (Get into the monitor from BASIC)  
BLOAD FTS      (Load the FTS program)
```

...Now insert the disk to be scanned...

```
900G           (Run the FTS program on this diskette)
```

The output might look like this...

```
T=12 S=0F  
T=13 S=0F  
T=14 S=0D  
T=14 S=0F
```

Here, only four possible files were found. **ZAP** should now be used to read track **\$12**, sector **\$0F**. At **+\$0C** is the track and sector of the first sector in the file. This sector can be read and examined to try to identify the file and its type. Usually a BASIC program can be identified, even though it is stored in tokenized form, from the text strings contained in the **PRINT** statements. An ASCII conversion chart (see page 8 in the *Apple II Reference Manual*) can be used to decode these character strings. Straight **T**-type files will also contain ASCII

text, with each line separated from the others with \$8D (carriage returns). B-type files are the hardest to identify, unless the address and length stored in the first 4 bytes are recognizable. If you cannot identify the file, assume it is Applesoft BASIC. If this assumption turns out to be incorrect, you can always go back and ZAP the file type in the CATALOG to try something else. Given below is an example ZAP to the CATALOG to create an entry for the file whose T/S List is at T=12 S=0F.

```
CALL -151
BLOAD ZAP
```

...insert disk to be ZAPped...

```
800:00 N 801<800.8FEM      (Zero sector area of memory)
80B:12 0F 02              (Track 12, Sector 0F, Type-A)
:C1 A0 A0 A0 A0 A0 A0     (Name is "A")
:A0 A0 A0 A0 A0 A0 A0     (fill name out with 29 blanks)
:A0 A0 A0 A0 A0 A0 A0
:A0 A0 A0 A0 A0 A0 A0
:A0 A0
02:11 0F 02 N 900G        (Write new sector image out as
                           first (and only) catalog sector)
```

The file should immediately be copied to another diskette and then the process repeated for each T/S List found by FTS until all of the files have been recovered. As each file is recovered, it may be RENAMED to its previous name. Once all the files have been copied to another disk, and successfully tested, the damaged disk may be re-INITIALIZED.

```
1 *****
2 *
3 * FTS: THIS PROGRAM SCANS THE *
4 * ENTIRE DISKETTE FOR WHAT *
5 * APPEAR TO BE TRACK/SECTOR *
6 * LISTS AND PRINTS THE *
7 * TRACK AND SECTOR OF EACH *
8 * ONE IT FINDS. *
9 *
10 * INPUT: NONE *
11 *
12 * ENTRY POINT: $900 *
13 *
14 * PROGRAMMER: DON D WORTH 2/15/81 *
15 *
16 *****

18 BELL EQU $87 BELL CHARACTER
19 RETURN EQU $8D CARRIAGE RETURN

21 * ZPAGE DEFINITIONS

23 PTR EQU $0 WORK POINTER
24 A1L EQU $3C MONITOR POINTER
```

	25	A2L	EQU	\$3E	MONITOR POINTER
	26	PREG	EQU	\$48	MONITOR STATUS REGISTER
	28	*	OTHER ADDRESSES		
	30	BUFFER	EQU	\$800	SECTOR DATA BUFFER
	31	LOCRLP	EQU	\$3E3	LOCATE RWTS PARMLIST SUBRTN
	32	RWTS	EQU	\$3D9	RWTS SUBROUTINE
	33	COUT	EQU	\$FDED	PRINT ONE CHAR SUBROUTINE
	34	PRBYTE	EQU	\$FDDA	PRINT ONE HEX BYTE SUBRTN
	36	*	RWTS PARMLIST DEFINITION		
	38		ORG	\$0	
0000: 00	39	RPLIOB	DS	1	IOB TYPE (\$01)
0001: 00	40	RPLSLT	DS	1	SLOT*16
0002: 00	41	RPLDRV	DS	1	DRIVE
0003: 00	42	RPLVOL	DS	1	VOLUME
0004: 00	43	RPLTRK	DS	1	TRACK
0005: 00	44	RPLSEC	DS	1	SECTOR
0006: 00 00	45	RPLDCT	DS	2	ADDRESS OF DCT
0008: 00 00	46	RPLBUF	DS	2	ADDRESS OF BUFFER
000A: 00 00	47	RPLSIZ	DS	2	SECTOR SIZE
000C: 00	48	RPLCMD	DS	1	COMMAND CODE
	49	RPLCNL	EQU	\$00	NULL COMMAND
	50	RPLCRD	EQU	\$01	READ COMMAND
	51	RPLCWR	EQU	\$02	WRITE COMMAND
	52	RPLCFM	EQU	\$04	FORMAT COMMAND
000D: 00	53	RPLRCD	DS	1	RETURN CODE
	54	RPLRWP	EQU	\$10	WRITE PROTECTED
	55	RPLRVM	EQU	\$20	VOLUME MISMATCH
	56	RPLRDE	EQU	\$40	DRIVE ERROR
	57	RPLRRE	EQU	\$80	READ ERROR
000E: 00	58	RPLTVL	DS	1	TRUE VOLUME
000F: 00	59	RPLPSL	DS	1	PREVIOUS SLOT
0010: 00	60	RPLPDR	DS	1	PREVIOUS DRIVE
	61		ORG	\$900	
	63	*	START TRACK/SECTOR JUST PAST DOS (TRACK 3)		
0900: 20 E3 03	65	FTS	JSR	LOCRLP	LOCATE RWTS PARMLIST
0903: 84 00	66		STY	PTR	AND SAVE POINTER
0905: 85 01	67		STA	PTR+1	
0907: A9 03	69		LDA	#3	FIRST NON-DOS TRACK
0909: A0 04	70		LDY	#RPLTRK	STORE IN RWTS LIST
090B: 91 00	71		STA	(PTR),Y	
090D: A0 08	73		LDY	#RPLBUF	
090F: A9 00	74		LDA	#<BUFFER	STORE BUFFER PTR IN LIST
0911: 91 00	75		STA	(PTR),Y	
0913: C8	76		INY		
0914: A9 08	77		LDA	#>BUFFER	
0916: 91 00	78		STA	(PTR),Y	
0918: A9 01	80		LDA	#RPLCRD	GET COMMAND CODE FOR READ
091A: A0 0C	81		LDY	#RPLCMD	AND STORE IN LIST
091C: 91 00	82		STA	(PTR),Y	
091E: A9 00	84		LDA	#0	ANY VOLUME WILL DO
0920: A0 03	85		LDY	#RPLVOL	
0922: 91 00	86		STA	(PTR),Y	
	88	*	NEW TRACK, START SECTOR AT ZERO		


```

0924: A0 05    90  NEWTRK  LDY  #RPLSEC
0926: A9 00    91          LDA  #0
0928: 91 00    92          STA  (PTR),Y

          94  *          NOW CALL RWTS TO READ THE SECTOR

092A: 20 E3 03  96  NEWSEC  JSR  LOCRPL      RELOAD POINTER TO PARMS
092D: 20 D9 03  97          JSR  RWTS        CALL RWTS
0930: A9 00    98          LDA  #0
0932: 85 48    99          STA  PREG        FIX P REG SO DOS IS HAPPY
0934: 90 26   100         BCC  SCAN        ALL IS WELL

          102 *          ERROR OCCURED, PRINT "RC=XX"

0936: 20 B3 09  104         JSR  PRTTS      PRINT TRACK/SECTOR
0939: A9 87   105         LDA  #BELL      BEEP THE SPEAKER
093B: 20 ED FD  106         JSR  COUT
093E: A9 52   107         LDA  #'R'      PRINT THE "RC="
0940: 20 ED FD  108         JSR  COUT
0943: A9 43   109         LDA  #'C'
0945: 20 ED FD  110         JSR  COUT
0948: A9 3D   111         LDA  #'='
094A: 20 ED FD  112         JSR  COUT
094D: A0 0D   113         LDY  #RPLRCD
094F: B1 00   114         LDA  (PTR),Y    GET RWTS RETURN CODE
0951: 20 DA FD  115         JSR  PRBYTE    PRINT RETURN CODE IN HEX
0954: A9 8D   116         LDA  #RETURN
0956: 20 ED FD  117         JSR  COUT
0959: 4C 8E 09  118         JMP  NXTSEC    GO ON

          120 *          NO ERROR, SEE IF SECTOR LOOKS LIKE A T/S LIST

095C: A2 00    122  SCAN    LDX  #0
095E: BD 00 08  123  SCLP0  LDA  BUFFER,X  MAKE SURE ITS NOT ALL ZERO
0961: D0 05    124          BNE  SCAN1
0963: E8      125          INX
0964: D0 F8    126          BNE  SCLP0
0966: F0 26    127          BEQ  NXTSEC    IF IT IS, SKIP IT

0968: A2 05    129  SCAN1  LDX  #5          START AT OFFSET 5
096A: BD 00 08  130  SCLP1  LDA  BUFFER,X
096D: D0 1F    131          BNE  NXTSEC    HEADER OF T/S MUST BE ZERO
096F: E8      132          INX
0970: E0 0C    133          CPX  #12        AT THE T/S PAIRS YET?
0972: 90 F6    134          BCC  SCLP1      NO, KEEP CHECKING

0974: BD 00 08  136  SCLP2  LDA  BUFFER,X  GET TRK
0977: C9 23    137          CMP  #35      MUST BE 0-34
0979: B0 13    138          BCS  NXTSEC
097B: E8      139          INX
097C: BD 00 08  140          LDA  BUFFER,X  GET SECTOR
097F: C9 10    141          CMP  #16      MUST BE 0-15
0981: B0 0B    142          BCS  NXTSEC
0983: E8      143          INX
0984: D0 EE    144          BNE  SCLP2

0986: 20 B3 09  146          JSR  PRTTS      ALL CONDITIONS MET
0989: A9 8D    147          LDA  #RETURN
098B: 20 ED FD  148          JSR  COUT

          150 *          BUMP SECTOR NUMBER OR TRACK AND CONTINUE

098E: A0 05    152  NXTSEC  LDY  #RPLSEC
0990: B1 00    153          LDA  (PTR),Y    GET LAST SECTOR
0992: 18      154          CLC

```

```

0993: 69 01      155      ADC    #1          BUMP BY ONE
0995: 91 00      156      STA    (PTR),Y      AND PUT IT BACK IN LIST
0997: C9 10      157      CMP    #16         TOO BIG?
0999: B0 03      158      BCS    NXTTRK
099B: 4C 2A 09    159      JMP    NEWSEC       NO, GO READ IT

099E: A0 04      161  NXTTRK  LDY    #RPLTRK
09A0: B1 00      162      LDA    (PTR),Y      GET LAST TRACK
09A2: 18          163      CLC
09A3: 69 01      164      ADC    #1          BUMP BY ONE
09A5: 91 00      165      STA    (PTR),Y      AND PUT IT BACK IN LIST
09A7: C9 11      166      CMP    #$11        CATALOG TRACK?
09A9: F0 F3      167      BEQ    NXTTRK       YES, SKIP OVER THAT ONE
09AB: C9 23      168      CMP    #35         DONE ALL 35 TRACKS?
09AD: B0 03      169      BCS    EXIT         YES, LEAVE
09AF: 4C 24 09   170      JMP    NEWTRK       NO, GO READ FIRST SECTOR
09B2: 60          171  EXIT    RTS

173  *          PRTTS: PRINT "T=XX S=XX"

09B3: A9 54      175  PRTTS   LDA    #'T'          PRINT "T"
09B5: 20 ED FD   176      JSR    COUT
09B8: A0 04      177      LDY    #RPLTRK
09BA: B1 00      178      LDA    (PTR),Y
09BC: 20 CC 09   179      JSR    PRTEQ       PRINT "=XX "

09BF: A9 53      181      LDA    #'S'          PRINT "S"
09C1: 20 ED FD   182      JSR    COUT
09C4: A0 05      183      LDY    #RPLSEC
09C6: B1 00      184      LDA    (PTR),Y
09C8: 20 CC 09   185      JSR    PRTEQ       PRINT "=XX "
09CB: 60          186      RTS

09CC: 48          188  PRTEQ   PHA
09CD: A9 3D      189      LDA    #'='
09CF: 20 ED FD   190      JSR    COUT
09D2: 68          191      PLA
09D3: 20 DA FD   192      JSR    PRBYTE
09D6: A9 20      193      LDA    #'
09D8: 20 ED FD   194      JSR    COUT
09DB: 60          195      RTS

```

--End assembly, 237 bytes, Errors: 0

COPY – CONVERT FILES

The **COPY** program demonstrates the use of the DOS File Manager subroutine package from assembly language. **COPY** will read as input a Binary type file, stripping off the address and length information, and write the data out as a newly created Text type file. The name of the input file is assumed to be “**INPUT**”, although this could just as easily have been inputted from the keyboard, and the name of the output file is “**OUTPUT**”. **COPY** is a single drive operation, using the last drive which was referenced.

To run **COPY**, load it and begin execution at **\$800**:

```
CALL -151      (Get into the monitor from BASIC)
BLOAD COPY    (Load the COPY program)
```

...Now insert the disk containing **INPUT**...

```
800G          (Run the COPY program)
```

When **COPY** finishes, it will return to **BASIC**. If any errors occur, the return code passed back from the File Manager will be printed. Consult the documentation on the File Manager parameter list in Chapter 6 for a list of these return codes.

```

1  *****
2  *
3  * COPY:THIS PROGRAM DEMONSTRATES *
4  * THE USE OF THE DOS FILE *
5  * MANAGER BY COPYING A BINARY *
6  * FILE TO A TEXT FILE. *
7  *
8  * INPUT: INPUT FILE NAME IS "INPUT" *
9  * OUTPUT FILE NAME IS "OUTPUT" *
10 *
11 * ENTRY POINT: $800 *
12 *
13 * PROGRAMMER: DON D WORTH 2/19/81 *
14 *
15 *****
17 BELL EQU $87 BELL CHARACTER

19 * ZPAGE DEFINITIONS

21 PTR EQU $0 WORK POINTER
22 BUFP EQU $2 BUFFER POINTER
23 EBYTE EQU $4
24 A1L EQU $3C MONITOR POINTER
25 A2L EQU $3E MONITOR POINTER

27 * OTHER ADDRESSES

29 BUFFER EQU $1000 DATA BUFFER
30 DOSWRM EQU $3D0 DOS WARMSTART ADDRESS
31 LOCRLP EQU $3E3 LOCATE RWTS PARMLIST SUBRTN

```

	32	LOCFPL	EQU	\$3DC	LOCATE FILE MGR PARMLIST SUB
	33	FM	EQU	\$3D6	FILE MANAGER ENTRY POINT
	34	COUT	EQU	\$FDED	PRINT ONE CHAR SUBROUTINE
	35	PRBYTE	EQU	\$FDDA	PRINT ONE HEX BYTE SUBRTN
	37	*		FILE MANAGER PARMLIST DEFINITION	
	39		ORG	\$0	
0000: 00	40	FMOCOD	DS	1	OPERATION CODE
	41	FMOCOP	EQU	\$01	OPEN
	42	FMOCCL	EQU	\$02	CLOSE
	43	FMOCRD	EQU	\$03	READ
	44	FMOCWR	EQU	\$04	WRITE
	45	FMOCDE	EQU	\$05	DELETE
	46	FMOCCL	EQU	\$06	CATALOG
	47	FMOCLO	EQU	\$07	LOCK
	48	FMOCUN	EQU	\$08	UNLOCK
	49	FMOCRE	EQU	\$09	RENAME
	50	FMOCPO	EQU	\$0A	POSITION
	51	FMOCIN	EQU	\$0B	INIT
	52	FMOCVE	EQU	\$0C	VERIFY
0001: 00	53	FMSBCD	DS	1	SUBCODE
	54	FMSBNO	EQU	\$00	NO OPERATION
	55	FMSBON	EQU	\$01	READ/WRITE ONE BYTE
	56	FMSBRA	EQU	\$02	READ/WRITE RANGE OF BYTES
	57	FMSBPO	EQU	\$03	POSITION AND DO ONE BYTE
	58	FMSBPR	EQU	\$04	POSITION AND DO RANGE
0002: 00 00 00	59	FMPRMS	DS	8	SPECIFIC PARAMETERS
0005: 00 00 00 00 00					
	61	*	OPEN	PARMS	
	62		ORG	FMPRMS	
0002: 00 00	63	FMRCCLN	DS	2	RECORD LENGTH
0004: 00	64	FMVOL	DS	1	VOLUME
0005: 00	65	FMDRV	DS	1	DRIVE
0006: 00	66	FMSLT	DS	1	SLOT
0007: 00	67	FMTYPE	DS	1	TYPE
	68	FMTYPT	EQU	0	TEXT
	69	FMTYPI	EQU	1	INTEGER
	70	FMTYPA	EQU	2	APPLESOFT
	71	FMTYPB	EQU	4	BINARY
0008: 00 00	72	FMNAME	DS	2	ADDRESS OF FILE NAME
	74	*	READ/WRITE	PARMS	
	75		ORG	FMPRMS	
0002: 00 00	76	FMRCNM	DS	2	RECORD NUMBER
0004: 00 00	77	FMOFFS	DS	2	BYTE OFFSET
0006: 00 00	78	FMRLN	DS	2	RANGE LENGTH
0008: 00 00	79	FMRAAD	DS	2	RANGE ADDRESS
	80	FMDATA	EQU	FMRAAD	DATA BYTE READ/WITTEN
	82	*	RENAME	PARMS	
	83		ORG	FMPRMS	
0002: 00 00	84	FMNNAM	DS	2	ADDRESS OF NEW NAME
	86	*	INIT	PARMS	
	87		ORG	FMPRMS	
	88	FMPAGE	EQU	FMSBCD	FIRST PAGE OF DOS IMAGE
	90	*	COMMON	PARMS	
	91		ORG	FMPRMS+8	
000A: 00	92	FMRC	DS	1	RETURN CODE
	93	FMRCOK	EQU	0	NO ERRORS
	94	FMRCBO	EQU	2	BAD OPCODE
	95	FMRCBS	EQU	3	BAD SUBCODE

```

          96 FMRCWP EQU 4 WRITE PROTECTED
          97 FMRCED EQU 5 END OF DATA
          98 FMRCNF EQU 6 FILE NOT FOUND
          99 FMRCBV EQU 7 BAD VOLUME
         100 FMRCIO EQU 8 I/O ERROR
         101 FMRCDF EQU 9 DISK FULL
         102 FMRCCLK EQU 10 FILE LOCKED
000B: 00 103 DS 1 NOT USED
000C: 00 00 104 FMFMWA DS 2 FILE MANAGER WORKAREA PTR
000E: 00 00 105 FMSTL DS 2 T/S LIST PTR
0010: 00 00 106 FMBUFF DS 2 DATA BUFFER PTR
         107 ORG $800

          109 * LOCATE FM PARMLIST

0800: 20 DC 03 111 COPY JSR LOCFPL FIND PARMLIST
0803: 84 00 112 STY PTR SET UP POINTER TO IT
0805: 85 01 113 STA PTR+1

          115 * OPEN INPUT FILE

0807: A0 08 117 LDY #FMNAME STORE INPUT FILE NAME
0809: A9 AC 118 LDA #<INAME PTR IN LIST
080B: 91 00 119 STA (PTR),Y
080D: C8 120 INY
080E: A9 09 121 LDA #>INAME
0810: 91 00 122 STA (PTR),Y
0812: A0 07 123 LDY #FMSTYPE BINARY FILE AS INPUT
0814: A9 04 124 LDA #FMSTYPB
0816: 91 00 125 STA (PTR),Y
0818: A2 01 126 LDX #1 OLD FILE EXPECTED
081A: 20 D3 08 127 JSR OPEN AND OPEN THE FILE
081D: 90 03 128 BCC INOP
081F: 4C BC 08 129 JMP ERROR ANY ERROR IS FATAL

0822: A5 02 131 INOP LDA BUFP
0824: 8D EA 09 132 STA IBUFP SAVE OPEN FILE BUFFER
0827: A5 03 133 LDA BUFP+1
0829: 8D EB 09 134 STA IBUFP+1
082C: 20 5A 09 135 JSR REWIND POSITION TO START OF FILE

          137 * OPEN OUTPUT FILE

082F: A0 08 139 LDY #FMNAME STORE OUTPUT FILE NAME
0831: A9 CA 140 LDA #<ONAME PTR IN LIST
0833: 91 00 141 STA (PTR),Y
0835: C8 142 INY
0836: A9 09 143 LDA #>ONAME
0838: 91 00 144 STA (PTR),Y
083A: A0 07 145 LDY #FMSTYPE TEXT FILE AS OUTPUT
083C: A9 00 146 LDA #FMSTYPT
083E: 91 00 147 STA (PTR),Y
0840: A2 00 148 LDX #0 NEW FILE IS OK
0842: 20 D3 08 149 JSR OPEN
0845: 90 0B 150 BCC OUTOP
0847: A0 0A 151 LDY #FMRC
0849: B1 00 152 LDA (PTR),Y
084B: C9 06 153 CMP #FMRCNF FILE NOT FOUND?
084D: F0 03 154 BEQ OUTOP YES, WAS ALLOCATED THEN
084F: 4C BC 08 155 JMP ERROR

0852: A5 02 157 OUTOP LDA BUFP SAVE OPEN OUTPUT FILE BUFFER
0854: 8D E8 09 158 STA OBUFP
0857: A5 03 159 LDA BUFP+1
0859: 8D E9 09 160 STA OBUFP+1

```

085C: 20 5A 09	161		JSR	REWIND	POSITION TO START OF FILE
	163	*		READ	ADDRESS/LENGTH FROM BINARY FILE
085F: A9 04	165		LDA	#4	READ 4 BYTES FIRST
0861: A0 06	166		LDY	#FMRALN	
0863: 91 00	167		STA	(PTR),Y	
0865: A9 00	168		LDA	#0	
0867: C8	169		INY		
0868: 91 00	170		STA	(PTR),Y	
086A: 20 74 09	171		JSR	READ	
	173	*		READ	ENTIRE BINARY FILE INTO MEMORY AT \$1000
086D: AD 02 10	175		LDA	BUFFER+2	COPY DATA LENGTH TO LIST
0870: A0 06	176		LDY	#FMRALN	
0872: 91 00	177		STA	(PTR),Y	
0874: AD 03 10	178		LDA	BUFFER+3	
0877: C8	179		INY		
0878: 91 00	180		STA	(PTR),Y	
087A: 18	181		CLC		
087B: AD 02 10	182		LDA	BUFFER+2	COMPUTE ENDING BYTE
087E: 48	183		PHA		
087F: 69 00	184		ADC	#<BUFFER	
0881: 85 04	185		STA	EBYTE	
0883: AD 03 10	186		LDA	BUFFER+3	
0886: 48	187		PHA		
0887: 69 10	188		ADC	#>BUFFER	
0889: 85 05	189		STA	EBYTE+1	
088B: 20 74 09	190		JSR	READ	READ BLOB INTO MEMORY
	192	*		WRITE	ENTIRE BLOB OUT INTO TEXT FILE
088E: A0 00	194		LDY	#0	
0890: 98	195		TYA		
0891: 91 04	196		STA	(EBYTE),Y	MARK END OF FILE
0893: 68	197		PLA		
0894: A0 07	198		LDY	#FMRALN+1	SET RANGE LENGTH
0896: 91 00	199		STA	(PTR),Y	
0898: 88	200		DEY		
0899: 68	201		PLA		
089A: 91 00	202		STA	(PTR),Y	
089C: 20 82 09	203		JSR	WRITE	WRITE BLOB FROM MEMORY
	205	*		WHEN	FINISHED, CLOSE FILES
089F: AD E8 09	207	EXIT	LDA	OBUFF	
08A2: 85 02	208		STA	BUFP	
08A4: AD E9 09	209		LDA	OBUFF+1	
08A7: 85 03	210		STA	BUFP+1	
08A9: 20 46 09	211		JSR	CLOSE	CLOSE OUTPUT FILE
08AC: AD EA 09	212		LDA	IBUFF	
08AF: 85 02	213		STA	BUFP	
08B1: AD EB 09	214		LDA	IBUFF+1	
08B4: 85 03	215		STA	BUFP+1	
08B6: 20 46 09	216		JSR	CLOSE	CLOSE INPUT FILE
08B9: 4C D0 03	217		JMP	DOSWRM	BACK TO DOS
	219	*		ERROR,	PRINT "ERRXX"
08BC: A0 0A	221	ERROR	LDY	#FMRC	FIND RETURN CODE
08BE: B1 00	222		LDA	(PTR),Y	
08C0: 48	223		PHA		
08C1: A9 45	224	ERR	LDA	#'E'	PRINT "ERR"
08C3: 20 ED FD	225		JSR	COUT	

```

08C6: A9 52      226      LDA    #'R'
08C8: 20 ED FD   227      JSR    COUT
08CB: 20 ED FD   228      JSR    COUT
08CE: 68         229      PLA
08CF: 20 DA FD   230      JSR    PRBYTE      PRINT HEX CODE
08D2: 00         231      BRK    DIE          HORRIBLY

                233      *      OPEN: COMPLETE PARMLIST AND OPEN FILE

08D3: AD D2 03   235      OPEN    LDA    DOSWRM+2    FIND DOS ENTRY
08D6: 85 03      236      STA    BUFP+1
08D8: A0 00      237      LDY    #0
08DA: 84 02      238      STY    BUFP          POINT AT BUFFER CHAIN

                240      *      SCAN DOS BUFFERS FOR A FREE ONE

08DC: B1 02      242      GBUF0    LDA    (BUFP),Y    LOCATE NEXT DOS BUFFER
08DE: 48         243      PHA
08DF: C8         244      INY
08E0: B1 02      245      LDA    (BUFP),Y
08E2: 85 03      246      STA    BUFP+1
08E4: 68         247      PLA
08E5: 85 02      248      STA    BUFP
08E7: D0 0A      249      BNE    GBUF          GOT ONE
08E9: A5 03      250      LDA    BUFP+1
08EB: D0 06      251      BNE    GBUF          GOT ONE

08ED: A9 0C      253      LDA    #12          NO FILE BUFFERS RETURN CODE
08EF: 48         254      PHA
08F0: 4C C1 08   255      JMP    ERR          GO PRINT MESSAGE

08F3: A0 00      257      GBUF    LDY    #0          LOOK AT FILENAME
08F5: B1 02      258      LDA    (BUFP),Y
08F7: F0 04      259      BEQ    GOTBUF        NONE THERE, FREE BUFFER
08F9: A0 24      260      LDY    #36          IT'S NOT FREE
08FB: D0 DF      261      BNE    GBUF0        GO GET NEXT ONE

08FD: A9 01      263      GOTBUF   LDA    #1
08FF: 91 02      264      STA    (BUFP),Y    MARK BUFFER IN USE

                266      *      FINISH COMPLETING OPEN LIST

0901: A0 00      268      LDY    #FMOCOD
0903: A9 01      269      LDA    #FMOCOP
0905: 91 00      270      STA    (PTR),Y    SET OPCODE TO OPEN
0907: A9 00      271      LDA    #0
0909: A0 02      272      LDY    #FMRCLN
090B: 91 00      273      STA    (PTR),Y    SET RECORD LENGTH TO 0
090D: C8         274      INY
090E: 91 00      275      STA    (PTR),Y
0910: A0 04      276      LDY    #FMVOL
0912: 91 00      277      STA    (PTR),Y    AND VOLUME (ANY VOL)

0914: 20 E3 03   279      JSR    LOCRLPL    FIND RWTS PARMS
0917: 84 3C      280      STY    A1L
0919: 85 3D      281      STA    A1L+1
091B: A0 01      282      LDY    #1
091D: B1 3C      283      LDA    (A1L),Y    GET SLOT*16
091F: 4A         284      LSR    A
0920: 4A         285      LSR    A
0921: 4A         286      LSR    A
0922: 4A         287      LSR    A          SLOT=SLOT/16
0923: A0 06      288      LDY    #FMSLT
0925: 91 00      289      STA    (PTR),Y    STORE IN LIST
0927: A0 02      290      LDY    #2

```

0929: B1 3C	291		LDA	(A1L),Y	GET DRIVE
092B: A0 05	292		LDY	#FMDRV	
092D: 91 00	293		STA	(PTR),Y	AND SLOT
	295	*	COMMON INTERFACE TO FILE MANAGER		
092F: A0 1E	297	CALLFM	LDY	#30	
0931: B1 02	298	CFMLP1	LDA	(BUFP),Y	GET THREE BUFFER PTRS
0933: 48	299		PHA		
0934: C8	300		INY		
0935: C0 24	301		CPY	#36	
0937: 90 F8	302		BCC	CFMLP1	
0939: A0 11	304		LDY	#FMBUFF+1	
093B: 68	305	CFMLP2	PLA		
093C: 91 00	306		STA	(PTR),Y	COPY THEM TO FM LIST
093E: 88	307		DEY		
093F: C0 0C	308		CPY	#FMFMWA	
0941: B0 F8	309		BCS	CFMLP2	
0943: 4C D6 03	311		JMP	FM	EXIT THRU FILE MANAGER
	313	*	CLOSE: CLOSE DOS FILE		
0946: A0 00	315	CLOSE	LDY	#FMOCOD	
0948: A9 02	316		LDA	#FMOCCL	
094A: 91 00	317		STA	(PTR),Y	
094C: 20 2F 09	318		JSR	CALLFM	CLOSE FILE
094F: 90 03	319		BCC	CLOCK	
0951: 4C BC 08	320		JMP	ERROR	
0954: A0 00	321	CLOCK	LDY	#0	FREE BUFFER
0956: 98	322		TYA		
0957: 91 02	323		STA	(BUFP),Y	
0959: 60	324		RTS		
	326	*	REWIND: POSITION TO START OF FILE		
095A: A0 02	328	REWIND	LDY	#FMRCNM	
095C: A9 00	329		LDA	#0	
095E: 91 00	330	REWLP	STA	(PTR),Y	ZERO RECORD NUMBER AND..
0960: C8	331		INY		
0961: C0 06	332		CPY	#FMOFFS+2	BYTE OFFSET.
0963: 90 F9	333		BCC	REWLP	
0965: A0 00	334		LDY	#FMOCOD	
0967: A9 0A	335		LDA	#FMOCPO	POSITION OPCODE
0969: 91 00	336		STA	(PTR),Y	
096B: 20 2F 09	337		JSR	CALLFM	EXIT VIA FILE MANAGER
096E: 90 03	338		BCC	REWRTS	CHECK FOR ERRORS
0970: 4C BC 08	339		JMP	ERROR	
0973: 60	340	REWRTS	RTS		
	342	*	READ: READ A RANGE OF BYTES TO \$1000		
0974: AD EA 09	344	READ	LDA	IBUFF	FIND PROPER BUFFER
0977: 85 02	345		STA	BUFP	
0979: AD EB 09	346		LDA	IBUFF+1	
097C: 85 03	347		STA	BUFP+1	
097E: A9 03	348		LDA	#FMOCRD	READ OPCODE
0980: D0 0C	349		BNE	DOIO	GO DO COMMON CODE
	351	*	WRITE: WRITE A RANGE OF BYTES FROM \$1000		
0982: AD E8 09	353	WRITE	LDA	OBUFP	FIND PROPER BUFFER
0985: 85 02	354		STA	BUFP	
0987: AD E9 09	355		LDA	OBUFP+1	


```

098A: 85 03      356      STA  BUFP+1
098C: A9 04      357      LDA  #FMOCWR      WRITE OPCODE
                                358      *      BNE  DOIO
                                360      *      DOIO: READ/WRITE A RANGE OF BYTES

098E: A0 00      362      DOIO  LDY  #FMOCOD
0990: 91 00      363      STA  (PTR),Y      SET OPCODE
0992: A0 01      364      LDY  #FMSBCD
0994: A9 02      365      LDA  #FMSBRA
0996: 91 00      366      STA  (PTR),Y      DO RANGE OF BYTES
0998: A0 08      367      LDY  #FMRAAD
099A: A9 00      368      LDA  #<BUFFER
099C: 91 00      369      STA  (PTR),Y      RANGE ADDRESS=$1000
099E: C8          370      INY
099F: A9 10      371      LDA  #>BUFFER
09A1: 91 00      372      STA  (PTR),Y
09A3: 20 2F 09   373      JSR  CALLFM      CALL FM TO DO I/O OPERATION
09A6: 90 03      374      BCC  DOIORT
09A8: 4C BC 08   375      JMP  ERROR
09AB: 60          376      DOIORT RTS
                                378      *      DATA

09AC: 49 4E 50   380      INAME  ASC  'INPUT
09AF: 55 54 20 20 20 20 20 20
09B7: 20 20 20 20 20 20 20 20
09BF: 20 20 20 20 20 20 20 20
09C7: 20 20 20
09CA: 4F 55 54   381      ONAME  ASC  'OUTPUT
09CD: 50 55 54 20 20 20 20 20
09D5: 20 20 20 20 20 20 20 20
09DD: 20 20 20 20 20 20 20 20
09E5: 20 20 20

09E8: 00 00      383      OBUFF  DS   2
09EA: 00 00      384      IBUFF  DS   2

--End assembly, 528 bytes, Errors: 0

```



DOS Ø.Ø1

Appendix B

Disk Protection Schemes

As the quantity and quality of Apple II software has increased, so has the incidence of illegal duplication of copyrighted software. To combat this, software vendors have introduced methods for protecting their software. Since most protection schemes involve a modified or custom Disk Operating System, it seems appropriate to discuss disk protection in general.

Typically, a protection scheme's purpose is to stop unauthorized duplication of the contents of the diskette, although it may also include, or be limited to, preventing the listing of the software (if it is in BASIC). This has been attempted in a variety of ways, all of which necessitate reading and writing non-standard formats on the disk. If the reader is unclear about how a normal diskette is formatted, refer to Chapter 3 for more information.

Early protection methods were primitive in comparison to what is being done now. Just as the methods of protection have improved, so have the techniques people have used to break them. The cycle seems endless. As new and more sophisticated schemes are developed, they are soon broken, prompting the software vendor to try to create even more sophisticated systems.

It seems reasonable at this time to say that it is impossible to protect a disk in such a way that it can't be broken. This is, in large part, due to the fact that the diskette must be "bootable"; i.e. that it must contain at least one sector (Track 0, Sector 0) which can be read by the program in the PROM on the disk controller card. This means that it is possible to trace the boot process by disassembling the normal sector or sectors that must be on the disk. It turns out that it is even possible to protect these sectors. Because of a lack of space on the PROM (256 bytes), the software doesn't fully check either the Address Field or the Data Field. But potential protection schemes which take advantage of this are limited and must involve only certain changes which will be discussed below.

Most protected disks use a modified version of Apple's DOS. This is a much easier task than writing one's own Disk Operating System and will be the primary area covered by this discussion.

Although there are a vast array of different protection schemes, they all consist of having some portion of the disk unreadable by a normal Disk Operating System. The two logical areas to alter are the Address Field and the Data Field. Each include a number of bytes which, if changed, will cause a sector to be unreadable. We will examine how that is done in some detail.

The Address Field normally starts with the bytes `$D5/$AA/$96`. If any one of these bytes were changed, DOS would not be able to locate that particular Address Field, causing an error. While all three bytes can and have been changed by various schemes, it is important to remember that they must be chosen in such a way as to guarantee their uniqueness. Apple's DOS does this by



Disk snooping

reserving the bytes `$D5` and `$AA`; i.e. these bytes are not used in the storage of data. The sequence chosen by the would-be disk protector can not occur anywhere else on the track, other than in another Address Field. Next comes the address information itself (volume, track, sector, and checksum). Some common techniques include changing the order of the information, doubling the sector numbers, or altering the checksum with some constant. Any of the above would cause an I/O error in a normal DOS. Finally, we have the two closing bytes (`$DE/$AA`), which are similar to the starting bytes, but with a difference. Their uniqueness is not critical, since DOS will read whatever

two bytes follow the information field, using them for verification, but not to locate the field itself.

The Data Field is quite similar to the Address Field in that its three parts correspond almost identically, as far as protection schemes are concerned. The Data Field starts with **\$D5/\$AA/\$AD**, only the third byte being different, and all that applies to the Address Field applies here also. Switching the third bytes between the two fields is an example of a protective measure. The data portion consists of 342 bytes of data, followed by a checksum byte. Quite often the data is written so that the checksum computation will be non-zero, causing an error. The closing bytes are identical to those of the Address Field (**\$DE/\$AA**).

As mentioned earlier, the PROM on the disk controller skips certain parts of both types of fields. In particular, neither trailing byte (**\$DE/\$AA**) is read or verified nor is the checksum tested, allowing these bytes to be modified even in track 0 sector 0. However, this protection is easily defeated by making slight modifications to DOS's RWTS routines, rendering it unreliable as a protective measure.

In the early days of disk protection, a single alteration was all that was needed to stop all but a few from copying the disk. Now, with more educated users and powerful utilities available, multiple schemes are quite commonly used. The first means of protection was probably that of hidden control characters imbedded in a file name. Now it is common to find a disk using multiple non-standard formats written even between tracks.

A state of the art protection scheme consists of two elements. First, the data is stored on the diskette in some non-standard way in order to make copying very difficult. Secondly, some portion of memory is utilized that will be altered upon a RESET. (For example, the primary text page or certain zero page locations) This is to prevent the software from being removed from memory intact.

Recently, several "nibble" or byte copy programs have become available. Unlike traditional copy programs which require the data to be in a predefined format, these utilities make as few assumptions as possible about the data structure. Ever since protected disks were first introduced, it has been asked, "why can't a track be read into memory and then written back out to another diskette in exactly the same way?". The problem lies with the self-sync or auto-sync bytes. (For a full discussion see Chapter 3) These bytes contain extra zero bits that are lost when read into memory. In memory it is impossible to determine the difference between a hexadecimal **\$FF** that was data and a hex **\$FF** that was a self-sync byte. Two solutions are currently being implemented in nibble copy programs. One is to analyze the data on a track with the hope

that the sync gaps can be located by deduction. This has a high probability of success if 13 or 16 sectors are present, even if they have been modified, but may not be effective in dealing with non-standard sectoring where sectors are larger than 256 bytes. In short, this method is effective but by no means foolproof. The second method is simple but likewise has a difficulty. It simply writes every hex **\$FF** found on the track as if it were a sync byte. This, however, will expand the physical space needed to write the track back out, since sync bytes require 25% more room. If enough hex **\$FF**s occur in the data, the track will overwrite itself. This can happen in general if the drive used to write the data is significantly slower than normal. Thus, we are back to having to analyze the data and, in effect, make some assumptions. It appears that, apart from using some hardware device to help find the sync bytes, a software program must make some assumptions about how the data is structured on the diskette.

The result of the introduction of nibble copy programs has been to “force the hand” of the software vendors. The initial response was to develop new protection schemes that defeated the nibble copy programs. More recent protection schemes, however, involve hardware and timing dependencies which require current nibble copy programs to rely heavily upon the user for direction. If the present trend continues, it is very likely that protection schemes will evolve to a point where automated techniques cannot be used to defeat them.



DOS 3.2 and DOS 3.3 meet ProDOS

Beneath Apple ProDOS

Contents

CHAPTER 1: INTRODUCTION.....	155
CHAPTER 2: TO BUILD A BETTER DOS	157
THE DEFICIENCIES OF DOS.....	157
ENTER PRODOS	158
MORE PRODOS ADVANTAGES	160
WHAT YOU GIVE UP WITH PRODOS.....	161
OTHER DIFFERENCES BETWEEN PRODOS AND DOS.....	163
CHAPTER 3: DISK II HARDWARE AND DISKETTE FORMATTING	165
TRACKS AND SECTORS	166
TRACK FORMATTING.....	167
ADDRESS FIELDS	174
DATA FIELDS	175
DISK II BLOCK AND SECTOR INTERLEAVING	176
INTRA-BLOCK INTERLEAVING.....	178
INTER-BLOCK INTERLEAVING.....	178
READING OR WRITING A BLOCK	179
READING OR WRITING CONSECUTIVE BLOCKS	180
CHAPTER 4: VOLUMES, DIRECTORIES, AND FILES.....	183
THE DISKETTE VOLUME	184
VOLUME OVERHEAD	185
VOLUME SPACE ALLOCATION – THE VOLUME BIT MAP	186
THE VOLUME DIRECTORY.....	187
THE VOLUME DIRECTORY HEADER.....	188
FILE DESCRIPTIVE ENTRIES	190
FILE STRUCTURES.....	194
FILE DATA TYPES.....	199
TXT FILES.....	200
BIN FILES	203
BAS FILES	204

OTHER FILE TYPES (VAR, REL, SYS).....	204
DIR FILES — PRODOS SUBDIRECTORIES	206
EMERGENCY REPAIRS.....	210
FRAGMENTATION	213
CHAPTER 5: THE STRUCTURE OF PRODOS.....	215
PRODOS MEMORY USE.....	215
GLOBAL PAGES.....	218
WHAT HAPPENS DURING BOOTING.....	220
CHAPTER 6: USING PRODOS FROM ASSEMBLY LANGUAGE.....	225
CAVEAT.....	225
DIRECT USE OF THE DISKETTE DRIVE	225
STEPPER PHASE OFF OR ON.....	227
MOTOR OFF OR ON.....	227
ENGAGE DRIVE 1 OR 2	228
READ A BYTE	228
SENSE WRITE PROTECT.....	228
WRITE LOAD AND WRITE A BYTE	228
CALLING A STORAGE DEVICE DRIVER (BLOCK ACCESS).....	229
CALLING THE DISK II DEVICE DRIVER	231
DEVICE DRIVER PARAMETER LISTS BY COMMAND CODE.....	232
CALLING THE MACHINE LANGUAGE INTERFACE	234
MLI PARAMETER LISTS BY FUNCTION CODE.....	237
MLI ERROR CODES	268
PASSING COMMAND LINES TO THE BASIC INTERPRETER	270
COMMON ALGORITHMS.....	272
IS PRODOS ACTIVE?.....	272
WHAT KIND OF MACHINE IS THIS?	272
HOW MUCH MEMORY IS IN THIS MACHINE?.....	272
GIVEN A PAGE NUMBER, SEE IF IT IS FREE	273
IS A BASIC PROGRAM RUNNING?.....	273
SETTING UP YOUR OWN RESET VECTOR.....	273
ACTIVATE A PRINTER OR OTHER PERIPHERAL	274
CHAPTER 7: CUSTOMIZING PRODOS	275
SYSTEM PROGRAMMING WITH PRODOS.....	275
INSTALLING A PROGRAM BETWEEN THE BI AND ITS BUFFERS.....	277
ADDING COMMANDS TO THE PRODOS BASIC INTERPRETER.....	278
DISABLE /RAM VOLUME FOR 128K MACHINES.....	279
WRITING YOUR OWN INTERPRETER.....	281
INSTALLING NEW PERIPHERAL DRIVERS.....	283
INSTALLING AN INTERRUPT HANDLER.....	285
DIRECT MODIFICATION OF PRODOS — A WORD OF WARNING	287
APPLYING PATCHES TO PRODOS	287

CHANGING THE NAME OF THE STARTUP FILE	290
PUT THE CURSOR ON COMMAND THAT CAUSED PRODOS ERROR.....	290
HOW TO WRITE TO A DIRECTORY FILE	290
CREATING A NEW FILE TYPE	291
RECOVERING DATA FROM A DAMAGED DISK.....	292
USING PRODOS WITH 40-TRACK DRIVES.....	292
FORCING PRODOS TO LOAD IN 48K	294
CHAPTER 8: PRODOS GLOBAL PAGES	295
BASIC INTERPRETER GLOBAL PAGE	296
PRODOS SYSTEM GLOBAL PAGE — MLI GLOBAL PAGE.....	299
APPENDIX A: EXAMPLE PROGRAMS.....	303
STORING THE PROGRAMS ON DISKETTE	304
APPENDIX B: DISKETTE PROTECTION SCHEMES.....	345
A BRIEF HISTORY OF APPLE SOFTWARE PROTECTION.....	345
PROTECTION METHODS.....	346
FORMAT ALTERATION	347
SIGNATURE.....	350
MEMORY PROTECTION.....	353
CODE PROTECTION	353
THE IDEAL PROTECTION SCHEME	354
NIBBLE COPIERS.....	354
HARDWARE BOARDS.....	354
FRONT DOOR METHOD.....	354
APPENDIX C: NIBBLIZING.....	357
ENCODING TECHNIQUES.....	357
“4 AND 4” ENCODING	358
“6 AND 2” ENCODING	359
THE ENCODING PROCESS	360
STAGE 1.....	367
STAGE 2.....	368
STAGE 3.....	369
APPENDIX D: THE LOGIC STATE SEQUENCER.....	371
LOGIC STATE SEQUENCER ROM.....	371
SEQUENCER EXAMPLE.....	377
APPENDIX E: PRODOS, DOS, AND SOS.....	379
CONVERTING FROM DOS TO PRODOS.....	379
WRITING PROGRAMS FOR PRODOS AND SOS.....	380

APPENDIX F: DIFFERENCES BETWEEN THE PRODOS 8 VERSIONS	383
CHANGES INTRODUCED IN THE 1.2 VERSION	383
RELOCATOR	383
MLI AND MI GLOBAL PAGE	384
QUIT CODE	385
CLOCK CODE	385
CHANGES INTRODUCED IN THE 1.3 VERSION	385
RELOCATOR	386
MLI	386
DISK II DEVICE DRIVER	386
BUGS IN VERSIONS 1.2 AND 1.3	386



Pardon me, sir, but do any of these look like your file?

Chapter 1

Introduction

Beneath Apple ProDOS is intended to serve as a companion to the manuals provided by Apple Computer, Inc. for ProDOS, providing additional information for the advanced programmer or for the novice Apple user who wants to know more about the structure of disks. It is not the intent of this manual to replace the documentation provided by Apple. Although, for the sake of continuity, some of the material covered in the Apple manuals is also covered here, it will be assumed that the reader is reasonably familiar with the contents of Apple's *ProDOS User's Manual* and *BASIC Programming With ProDOS*. Since all chapters presented here may not be of use to each Apple owner, each has been written to stand on its own. Readers of our earlier book, *Beneath Apple DOS*, will notice that we have retained the basic organization of that book in an attempt to help them familiarize themselves with *Beneath Apple ProDOS* more quickly.

The information presented here is a result of intensive disassembly and annotation of various versions of ProDOS by the authors. It also uses as a reference various application notes and preliminary documentation from Apple. Although no guarantee can be made concerning the accuracy of the information presented here, all of the material included in *Beneath Apple ProDOS* has been thoroughly researched and tested.

There were several reasons for writing *Beneath Apple ProDOS*:

- To show how to access ProDOS and/or the Disk II drive directly from machine language.
- To help you fix damaged disks.
- To correct errors and omissions in the Apple documentation.
- To allow you to customize ProDOS to fit your needs.
- To provide complete information on the diskette formatting.
- To document the internal logic of ProDOS.
- To present a critical, non-Apple perspective of ProDOS.
- To provide more examples of ProDOS programming.
- To help you learn about how an operating system works.

When Apple introduced ProDOS Version 1.0.1 in January 1984, three manuals were available: the *ProDOS User's Manual* documents the use of ProDOS utilities; the *BASIC Programming With ProDOS* manual describes the command language supported by the BASIC Interpreter and how to write BASIC programs which access the disk; and the *ProDOS Technical Reference Manual (for the Apple II family)* documents the assembly language interfaces to ProDOS. It should be stated that this technical reference manual represents the best internal documentation Apple has ever provided to users of one of their operating systems.

Unfortunately, the *ProDOS Technical Reference Manual* documents a prerelease version of ProDOS, and is not entirely accurate for the current release at the time of this writing. In addition, many sections require further explanation before the interfaces they describe can be used at all. For example, the discussion of how one adds a command to the BASIC interpreter omits several vital pieces of information which are documented fully in *Beneath Apple ProDOS*. In addition, none of the Apple documentation addresses diskette formatting or direct access of the Disk II family of controllers from assembly language. *Beneath Apple ProDOS* was written in an attempt to improve upon the documentation base established by Apple. Most of the topics covered by Apple's technical manual are covered here also, but they are explained in a different and, we hope, clearer way, based upon a programmer's understanding of the code in the ProDOS Kernel and the BASIC Interpreter.

We have also added substantial information on diskette formatting and repair, the internal logic and structure of ProDOS, and customizing techniques, as well as providing several example programs and quick reference materials.

In addition to the ProDOS specific information provided, many of the discussions also apply to other operating systems in the Apple II and Apple III family of machines. For example, disk formatting at the track and sector level is for the most part the same. Also, the format of a ProDOS volume is nearly identical to that of an Apple III SOS volume.

Chapter 2

To Build a Better DOS

From June 1978 to January 1984, the primary disk operating system for the Apple II family was Apple DOS. Throughout its first six years of existence, DOS has gone through a number of changes, culminating in its final version, DOS 3.3. DOS was originally designed primarily to support the BASIC programmer, but has since been adopted by assembly language programmers and by the majority of Apple users for a variety of applications.

THE DEFICIENCIES OF DOS

Although it is a flexible and easy to use operating system, DOS suffers from many weaknesses. Among these are:

- **DOS is slow.** Since each byte read from the disk is copied between memory buffers up to three times, a large portion of the actual overhead in reading data from the disk is in processor manipulation after the data has been read. To circumvent this, several “fast DOS” packages have been marketed by third parties which heavily modify DOS to prevent multiple buffering under certain circumstances.
- **DOS is device dependent.** When DOS was developed, the only mass storage device for the Apple was the Disk II diskette drive. Now that diskette drives with increased capacity and hard disks are available, a more device independent file organization is needed. DOS is limited in the number of files which can be stored on a diskette as well as their maximum size. These are significant drawbacks when a hard disk with five million bytes or more is used.
- Over the years, new hardware has been introduced by Apple and other manufacturers which DOS does not intrinsically support. The Apple IIe with its 80-column card and the Thunderclock are examples.
- **DOS is difficult to customize.** There are few external “hooks” provided to allow system programmers the opportunity to personalize the operating

system to special applications. For example, a new command cannot be added to DOS without version dependent patches.

- DOS file structures and system calls are **incompatible** with other operating systems. Each operating system Apple has announced in the past has had its own way of organizing data on a diskette. There is no compatibility between DOS, SOS and the Apple Pascal system. This means that special utilities must be written to move data between these systems and that applications developed in one environment will not run without major modifications under any other system.
- DOS does not provide a consistent mechanism for supporting multiple peripherals which can generate hardware **interrupts**. In the past, various manufacturers have implemented interrupt handlers on their own, often resulting in incompatibilities between their devices.
- DOS provides little standardization of **memory use** and of operating system interfaces. Most “interesting” locations within DOS are internalized and therefore not officially available to the programmer. Also, since there is no standard way to set aside portions of memory for specific applications, it is difficult to put a program in a “safe” place so that it may co-reside with another application.
- Although DOS allows most of its commands to be executed from within a BASIC program, additional function is needed. Under DOS there is no way to conveniently read a file directory from a BASIC program, or to save and restore Applesoft’s variables, for example. Likewise, the implementation of program **CHAINing** is not integrated into DOS.
- Additional functions under DOS which would be desirable (to name only a few) are: a display of the amount of freespace left on a diskette; a way to show the address and length parameters stored with a binary file; and a way to create unbootable data disks to increase storage space for user files.

ENTER PRODOS

In January 1984, Apple introduced a new disk operating system for its Apple II family of computers. ProDOS is intended to replace DOS 3.3 as the standard Apple II operating system, and it is now being shipped with all new Disk II drives instead of DOS. Although, on the surface, ProDOS is very similar in appearance to DOS 3.3, it represents a major redesign and is a new and separate system. From the beginning, ProDOS addresses all of DOS’s weaknesses mentioned above:

- ProDOS is up to **eight times faster** than DOS in disk access. A new “direct read” mode has been implemented which allows multisector reads to be performed directly from the disk to the programmer’s buffer without multiple buffering within ProDOS itself. When performing direct reads, ProDOS can transfer data from the diskette at a rate of eight kilobytes per second (at best, DOS can read one kilobyte per second). Even when reading small amounts of data from the disk, ProDOS does less multiple buffering than does DOS.
- ProDOS provides a **device independent** interface to “foreign” mass storage devices. The concept of a hierarchically organized disk “volume” was created to allow for large capacity devices, and vectors are provided to allow device drivers for non-standard disks to be integrated into ProDOS. Directories may be dynamically expanded to unlimited size to allow for large numbers of files, and an individual file may now occupy up to 16 million bytes of space on a volume. The largest volume which can be supported is 32 million bytes.
- Device **driver support** has also been provided for calendar/clock peripherals, allowing time and date stamping of files, and support for the Apple IIe and IIc 80-column hardware is a part of ProDOS.
- Learning from its mistakes with DOS, Apple has externalized as many ProDOS functions as possible through well defined **system calls**. In addition to standard file management system calls, interfaces are provided to support user written commands to the BASIC interpreter, and to invoke a ProDOS command from within an assembly language program.
- The ProDOS file and volume structure is nearly identical to that of the Apple III SOS operating system. There are even strong similarities between ProDOS system calls and those on Apple’s Macintosh! A ProDOS volume may be accessed from SOS directly without the need for a special utility program. ProDOS system calls are a large subset of those offered under SOS and applications may be developed which will easily port between the two operating systems.
- ProDOS defines a protocol which **interrupting** devices may use to coexist harmoniously in the same machine. Up to four interrupt drivers may be installed in ProDOS, and each device need not know that the others exist.
- Most locations of general interest have been placed in externally accessible areas of memory called **global pages**. Through a global page, a user written program can obtain the current ProDOS version number, the most recent values entered on a ProDOS command line, or the configuration of the current hardware including the machine type, memory size, and

the contents of the peripheral card slots. In addition, a voluntary system has been provided to “fence off” portions of memory for special uses by marking a memory bit map in the system global page.

- New support has been provided under ProDOS for BASIC programmers. A BASIC program can now read a directory file, make a “snapshot” of its variables on disk and later restore them, and chain between programs, preserving variables.
- The **CATALOG** command under ProDOS displays the address and length values of binary files as well as the space remaining on a disk volume.

MORE PRODOS ADVANTAGES

In addition to addressing needs which grew out of DOS, Apple has also come up with **other enhancements** with ProDOS:

- A new “smart” **RUN** command (“-”) has been added which will automatically perform the function of a **RUN**, **EXEC**, or **BRUN** as appropriate depending upon the type of file being **RUN**.
- The assembly language interface has been expanded to include obtaining and updating statistical information about a file, moving the end of file mark in a file, allowing line at a time reads versus byte stream reads, determining the names of diskettes mounted in online drives, and creating new files or directories. In addition, entry points are included to allow applications to pass control from program to program and to allocate memory.
- The language independent, file management portion of ProDOS (the Kernel) is a separate unit from the BASIC support routines. Applications may be written which reclaim the memory normally occupied by BASIC support routines.
- All ProDOS utilities are menu oriented with enhanced user interfaces.
- Owners of the Extended 80-column card in an Apple IIe have access to a 64K “RAM/electronic disk drive” under ProDOS. Data stored there may be accessed almost instantaneously allowing much more efficient loading and storing of programs and data.
- Applesoft string “garbage collection” has been rewritten under ProDOS, and is now many times faster and more efficient.
- Files may be restricted or “locked” by type of access. Read only files may be established, or files which may be written but not destroyed, for example.

- The binary save (**BSAVE**) command has been enhanced under ProDOS. **BSAVE**s into existing binary files whose A and/or L keywords are omitted will use the current values of the target file. Also, other file types besides **BIN** files may be **BLOADEd** and **BSAVED**, allowing direct modification at a byte-by-byte level. (For example, one can **BLOAD** a text file and examine it in memory, making modifications to the hex image.)
- The record length of a random access text file is now stored with the file, allowing subsequent BASIC programs to access it without knowing its record length.
- Data disk volumes may now be created which do not contain an image of the operating system. ProDOS makes more efficient use of the disk, resulting in slightly more user storage for files.
- More information about a file is stored in the directory entry under ProDOS than under DOS. The length of a binary or Applesoft file, for example, is stored in the directory, not in the file itself.
- The manner in which the ProDOS BASIC Interpreter intercepts a BASIC program's command lines has been improved and is more reliable. It is now very difficult to "disconnect" ProDOS as could occur under DOS.
- More file types (256) are available under ProDOS. Some are "user definable."

WHAT YOU GIVE UP WITH PRODOS

ProDOS is not for everyone, however. There are a number of **disadvantages** to moving from DOS to ProDOS:

- Most assembly language programs which ran under DOS will have to be rewritten for ProDOS. The file management interfaces are completely different, and the "**PRINT control-D**" mechanism which worked from assembly language under DOS no longer works under ProDOS. This means that most commercial applications, such as word processors, compilers, and spreadsheets, will not be available for ProDOS until they are converted. This state of affairs will change, however, since ProDOS is now the "official" operating system for Apple II computers.
- Apple's older version of BASIC, Integer BASIC, is not supported under ProDOS. Indeed, Applesoft must be in the motherboard ROMs for the ProDOS BASIC Interpreter to work at all. This means that only the ProDOS Kernel, used in a standalone, run-time environment, will run on

an original, Integer Apple II. It is likely that someone (probably not Apple) will soon market an Integer BASIC interpreter for ProDOS, however.

- ProDOS requires 64K to support BASIC programming and commands. It can be made to run in 48K for run-time assembly language applications, but 64K is required to run the BASIC Interpreter which incorporates all of the ProDOS commands (e.g., **CATALOG**, **BLOAD**, etc.).
- Under BASIC, less memory is available to the program. Under DOS, **HIMEM** was set at \$9600 with three file buffers built into DOS. Under ProDOS, **HIMEM** is at \$9600 with no file buffers built in. Thus, as soon as a ProDOS BASIC program opens a file, **HIMEM** is moved down and 1K less memory is available. Likewise, since the Kernel occupies the Language Card (or bank switched memory), this space may not be used for other purposes. (DOS could be relocated into the language card to make more space available to BASIC programs. Also, Applesoft enhancement aid programs typically were loaded into the language card's alternate 4K bank under DOS. This is where ProDOS stores its **QUIT** code now).
- ProDOS only maintains a single directory prefix for all volumes, rather than remembering a default prefix for each volume. Hence, diskette swapping and access to multiple volumes at once can be cumbersome.
- Although the pathname for a file may be 64 characters, the actual name of a file may be only 15 characters, and may not include any special characters or blanks (other than "period"). 30 characters were permitted under DOS.
- Under DOS, up to 16 files may be opened concurrently by a BASIC program. Under ProDOS, only eight files may be opened at once. Also, an open file "cost" 595 bytes under DOS; under ProDOS, a 1024-byte buffer is allocated.
- BASIC programs which are computationally oriented will run about four percent slower on ProDOS than they did under DOS. This is because the ProDOS BASIC Interpreter leaves Applesoft **TRACE** running (invisibly) at all times so that it can monitor the execution of the program and perform garbage collection and disk commands. On the other hand, if strings or disk accesses are used, this degradation of performance will be more than offset by improvements in these areas.
- Several DOS commands have been removed, including **NOMON**, **MON**, and **VERIFY**. There is no way to see the commands in an **EXEC** file as they are executed.
- If a ProDOS directory is destroyed, it is harder to reconstruct than was the DOS **CATALOG** track. More information is stored in the directory making it harder to identify a file's type by examining its data blocks. Also, since

seedling files do not have index blocks (similar to DOS Track/Sector Lists), they are almost impossible to find once their directory entries are gone.

OTHER DIFFERENCES BETWEEN PRODOS AND DOS

There are a few other minor differences between ProDOS and DOS which are worth noting:

- The **BRUN** command now calls the target program rather than jumping to it as did DOS. The invoked program may return to ProDOS via a return subroutine.
- **CLOSE** will not produce an error message if the file named is not currently open.
- **APPEND** implies **WRITE**. It is not necessary to follow an **APPEND** command with a **WRITE** command in a BASIC program.
- ASCII text in ProDOS directory entries or **TEXT** files is stored with the most significant bit off.



The original disk formatters

Chapter 3

Disk II Hardware and Diskette Formatting

This chapter will explain how data is stored on a floppy diskette using a disk drive (Disk II family or equivalent). Much of the information in this chapter is applicable not only to ProDOS, but also to other operating systems on the Apple computer (DOS, Pascal, CP/M). Because ProDOS isolates device specific code, the contents of this chapter should not be considered a prerequisite for understanding succeeding chapters.

For system housekeeping, ProDOS divides external storage devices into blocks. Each **block** contains 512 bytes of information. It is device independent in that each device has its own driver. This driver enables ProDOS to read and write blocks, and additionally to obtain the status of a device. The device itself may actually store information in a number of ways and not necessarily in blocks. Blocks can be thought of as a conceptual unit of data that was created in software, having little or no relation to how data is actually stored on an external storage device. In fact, the standard Disk II stores information in a track and sector format. The device driver provides a mapping between these tracks and sectors, and the blocks. Since a sector contains 256 bytes, two **sectors** are required for each block. There are 560 sectors on a diskette and therefore 280 blocks. Chapter 4 deals with how ProDOS allocates these blocks to create files.

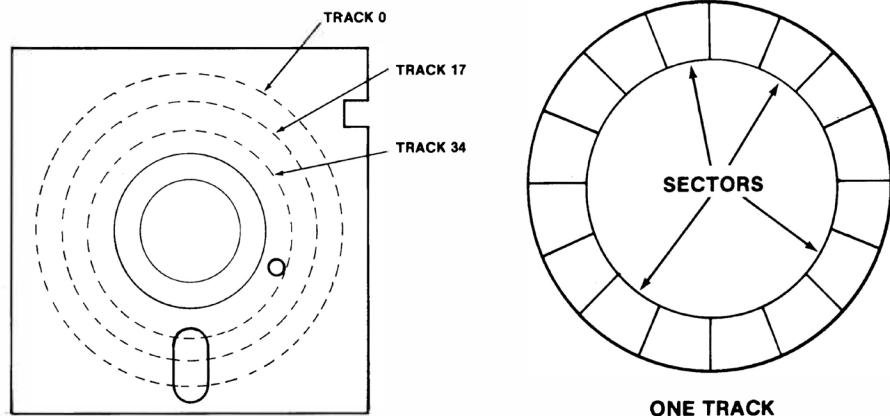


Figure 3.1 Tracks and Sectors

TRACKS AND SECTORS

As stated above, a diskette is divided into tracks and sectors. This is done during the initialization or formatting process. A **track** is a physically defined circular path which is concentric with the hole in the center of the diskette. Each track is identified by its distance from the center of the disk. Similar to a phonograph stylus, the read/write head of the disk drive may be positioned over any given track. The tracks are similar to the grooves in a record, but they are not connected in a spiral. Much like playing a record, the diskette is spun at a constant speed while the data is read from or written to its surface with the read/write head. Apple formats its diskettes into 35 tracks, numbered from 0 to 34, track 0 being the outermost track and track 34 the innermost. Figure 3.1 illustrates the concept of tracks, although they are invisible to the eye on a real diskette.

It should be pointed out, for the sake of accuracy, that the disk arm can position itself over 70 distinct locations or **phases**. To move the arm from one track to the next, two phases of the stepper motor which moves the arm must be cycled. This implies that data might be stored on 70 tracks, rather than 35. Unfortunately, the resolution of the read/write head is such that attempts to use these phantom **half** tracks create so much cross-talk that data is lost or overwritten. Although standard ProDOS uses only full tracks (even phases), some copy protected disks use half tracks (odd phases) or combinations of the two. This will work provided that no data is closer than two phases from other data. See Appendix B for more information on copy protection schemes.

A sector is a subdivision of a track. It is the smallest unit of “updatable” data on the diskette. While ProDOS reads or writes data a block at a time (two sectors), the device driver operates on one sector at a time. This allows the device driver to use only a small portion of memory as a buffer during read or write operations. Apple has used two different track formats to date. The initial operating system divided the track into 13 sectors but all recent operating systems use 16 sectors. The sectoring does not use the index hole, provided on most diskettes, to locate the first sector of the track. The implication is that the software must be able to locate any given track and sector with no help from the hardware. This scheme, known as **soft sectoring**, takes a little more space for storage but allows flexibility, as evidenced by the previous change from 13 sectors to 16 sectors per track. The following table categorizes the amount of data stored on a diskette under ProDOS. Both system and data diskettes are categorized.

DISKETTE ORGANIZATION

Tracks	35
Sectors Per Track	16
Sectors Per Diskette	560
Bytes Per Sector	256
Bytes Per Diskette.....	143,360

Usable* Blocks For Data Storage

ProDOS System Diskette.....	221
ProDOS Data Diskette.....	273

Usable* Bytes Per Diskette

ProDOS System Diskette.....	113,152
ProDOS Data Diskette.....	139,776

*System Diskette includes ProDOS and BASIC.SYSTEM files only.

TRACK FORMATTING

Up to this point we have broken down the structure of data to the track and sector level. To better understand how data is stored and retrieved, we will start at the bottom and work up.

As this manual is about software (ProDOS), we will deal primarily with the function of hardware rather than explain how it performs that function. For example, while data is in fact stored as a continuous stream of analog signals, we will deal with discrete **digital** data, i.e., a “0” or a “1”. We recognize that the hardware converts analog data to digital data, but how this is accomplished is beyond the scope of this manual. For a full and detailed explanation of the hardware, please refer to Jim Sather’s excellent book, *Understanding the Apple II*, published by Quality Software.

Data bits are recorded on the diskette in precise intervals. The hardware recognizes each of these intervals as either a “0” or a “1”. We will define these intervals to be **bit cells**. A bit cell can be thought of as the distance the diskette moves in four machine cycles, which is about four microseconds. Using this representation, data written on and read back from the diskette takes the form shown in Figure 3.2. The data pattern shown represents a binary value of 101.

A byte as recorded on the disk consists of eight (8) consecutive bit cells. The **most significant bit** cell is usually referred to as bit cell 7, and the **least significant** is bit cell 0. When reference is made to a specific data bit (i.e. data bit 5), it is with respect to the corresponding bit cell (bit cell 5). Data is written and read serially, one bit at a time. Thus, during a write operation, bit cell 7 of

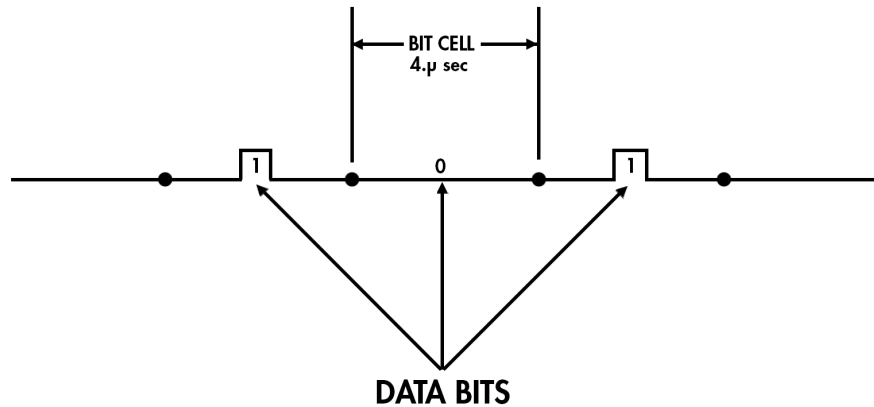


Figure 3.2 Bits on Diskette

each byte is written first, and bit cell 0 is written last. Correspondingly, when data is being read back from the diskette, bit cell 7 is read first and bit cell 0 is read last. Figure 3.3 illustrates the relationship of the bits within a byte.

To graphically show how bits are stored and retrieved, we must take certain liberties. The diagrams are a representation of what functionally occurs within the disk drive. For the purposes of our presentation, the hardware interface to the diskette will be represented as an 8-bit **data register**. Since the hardware involved considerably more complication, from a software standpoint, it is reasonable to use the data register, as it accurately embodies the function of data flow to and from the diskette. For a further discussion of the hardware, please see Appendix D.

Figure 3.4 shows the three bits, 101, being read from the diskette data

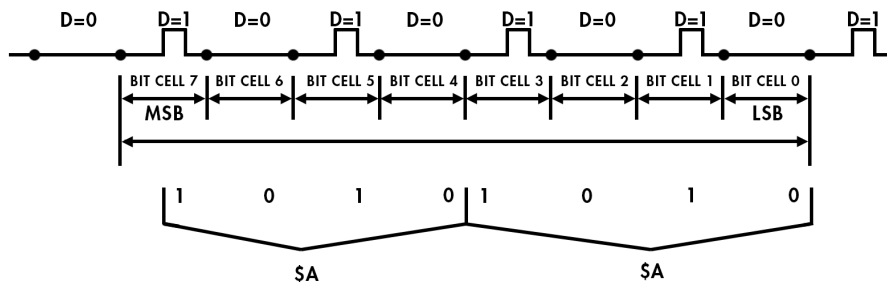


Figure 3.3 One Byte on Diskette

stream into the data register. Of course, another five bits would be read to fill the register.

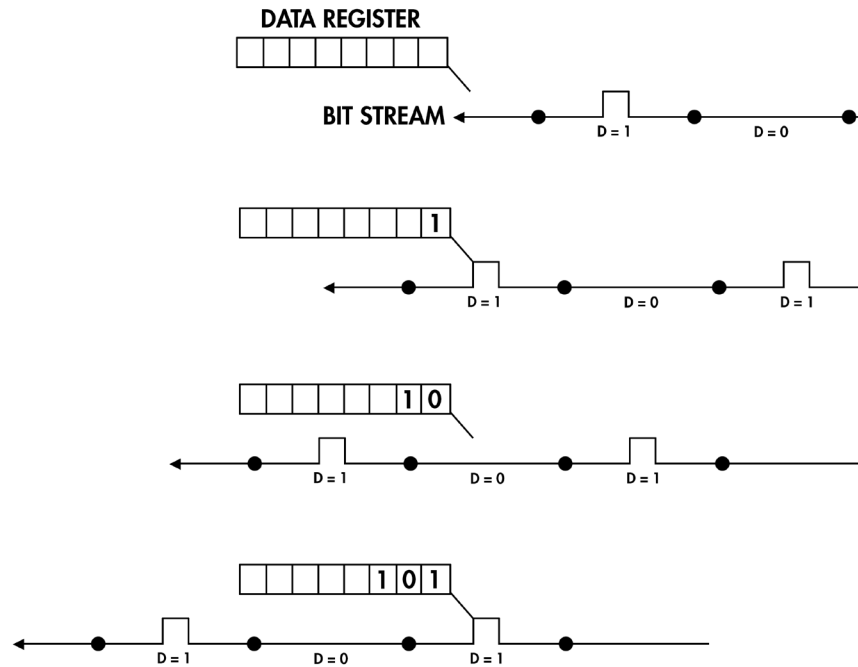


Figure 3.4 Reading Data from a Diskette

Writing data can be depicted much in the same way (See Figure 3.5). It should be noted that, while in write mode, zeroes are being brought into the data register to replace the data being written. It is the task of the software to make sure that the register is loaded and instructed to write on 32-cycle (microsecond) intervals. If not, zero bits will continue to be written every four cycles, which is in fact exactly how self-sync bytes are created. Self sync bytes will be covered in detail shortly.

A **field** is made up of a group of consecutive bytes. The number of bytes varies, depending upon the nature of the field. The two types of fields present on a diskette are the **Address Field** and the **Data Field**. They are similar in that they both contain a prologue, a data area, a checksum, and an epilogue. Each field on track is separated from adjacent fields by a number of bytes. These areas of separation are called **gaps** and are provided for two reasons. First, they allow the updating of one field without affecting adjacent fields (on the Apple, only data fields are updated). Secondly, they allow the computer time to decode the address field before the corresponding data field can pass beneath the read/write head.

All gaps are primarily alike in content, consisting of self-sync hexadecimal **FFs** and vary only in the number of bytes they contain. Figure 3.6 is a diagram of a portion of a typical track, broken into its major components.



Figure 3.5 Writing Data to a Diskette

Self sync or auto-sync bytes are special bytes that make up the three different types of gaps on a track. They are so named because of their ability to automatically bring the hardware into synchronization with data bytes on the disk. The difficulty in doing this lies in the fact that the hardware reads bits, and the data must be stored as 8-bit bytes. It has been mentioned that a track is literally a continuous stream of data bits. In fact, at the bit level, there is no way to determine where a byte starts or ends, because each bit cell is exactly the same, written in precise intervals with its neighbors. When the drive is instructed to read data, it will start wherever it happens to be on a particular track. That could be anywhere among the 50,000 or so bits on a track. The hardware finds the first bit cell with data in it and proceeds to read the following seven bits of data into the 8-bit register. In effect, it assumes that it has started at the beginning of a data byte. Of course, in reality, it could have started at any of the “1” bits of the byte. Pictured in Figure 3.7 is a small portion of a track.

From looking at the data, there is no way to tell what bytes are represented, because we don’t know where to start. This is exactly the problem that self-sync bytes overcome.

TRACK FORMAT

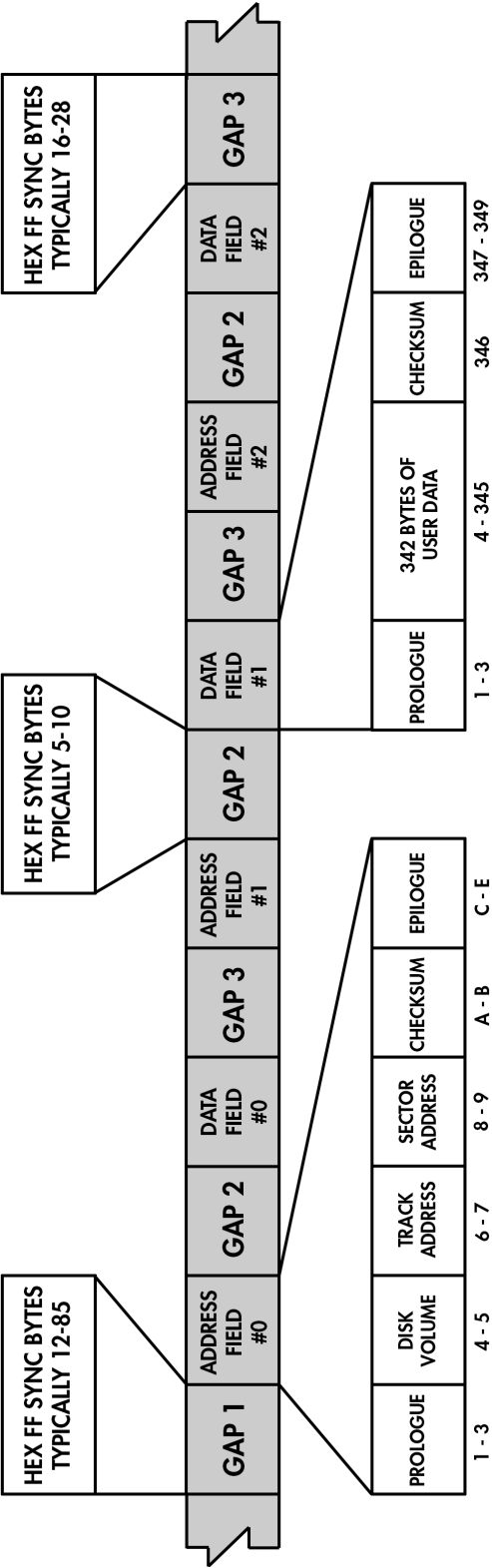


Figure 3.6 Track Format

A **self-sync** byte is defined to be a hexadecimal **FF** with a special difference. It is, in fact, a 10-bit byte rather than an 8-bit byte. Its two extra bits are zeroes. Figure 3.8 shows the difference between a normal data hex **FF** that might be found elsewhere on the disk and a self-sync hex **FF** byte.

A self-sync byte is generated by using a 40-cycle (microsecond) loop while writing an **FF**. A bit is written every four cycles, so two of the zero bits brought into the data register while the **FF** was being written are also written to the disk, making the 10-bit byte. It can be shown, using Figure 3.9, that four self-sync bytes are sufficient to guarantee that the hardware is reading valid data. The reason for this is that the hardware requires the first bit of a byte to be a “1”. Pictured at the top of the figure is a stream of four self-sync bytes followed by a normal **FF**. Each row below that demonstrates what the hardware will read should it start reading at any given bit in the first byte. In each case, by the time the four sync bytes have passed beneath the read/write head, the hardware will be synced to read the data bytes that follow. As long as the disk is left in read mode, it will continue to correctly interpret the data unless there is an error on the track.

0 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 1 1 1 0 1 1 0 1 1 1 0 1 0 1

Figure 3.7 An Example Bit Stream on a Diskette

We can now discuss the particular portions of a track in detail. The three gaps will be covered first. Unlike some other disk formats, the size of the three gap types will vary from drive to drive and even from track to track. During the formatting process, ProDOS will start with large gaps and keep making them smaller until an entire track can be written without overlapping itself. A minimum number of self-sync bytes is maintained for each gap type. The result is fairly uniform gap sizes within each particular track.

Gap 1 is the first data written to a track during initialization. Its purpose is twofold. The gap originally consists of 128 self-sync bytes, a large enough area to insure that all portions of a track will contain data. Since the speed of a particular drive may vary, the total length of the track in bytes is uncertain, and the percentage occupied by data is unknown. The initialization process is set up, however, so that even on drives of differing speeds, the last data field written will overlap Gap 1, providing continuity over the entire physical track.

NORMAL BYTE HEX FF	SELF-SYNC BYTE HEX FF
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 0 0

Figure 3.8 Comparison of Normal Byte and Self-Sync Byte

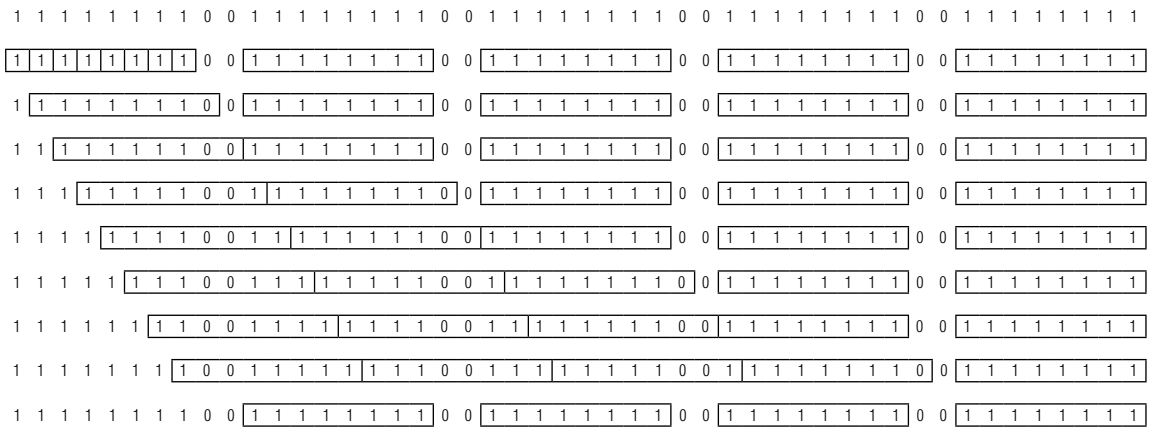


Figure 3.9 Self-Sync Bytes

Unlike earlier operating systems, ProDOS will let you know if your drive is too fast or too slow. The remaining portion of Gap 1 must be approximately 75% as long as a Gap 3 on that track, enabling it to serve as a Gap 3 type for Address Field number 0 (See Figure 3.6 for clarity).

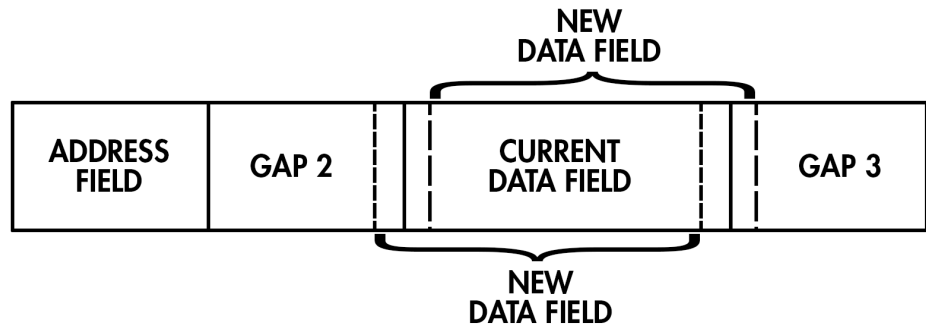


Figure 3.10 ProDOS Doesn't Always Write in the Same Place

Gap 2 appears after each Address Field and before each Data Field. Its primary purpose is to provide time for the information in an Address Field to be decoded by the computer before a read or write takes place. If the gap was too short, the beginning of the Data Field might spin past while ProDOS was still determining if this was the sector to be read. The 200 cycles that five self-sync bytes provide seems ample time to decode an Address Field. When a Data Field is written, there is no guarantee that the write will occur in exactly the same spot each time. This is due to the fact that the drive which is rewriting the Data Field may not be the one which originally formatted or wrote it. Since the speed of the drives can vary, it is possible that the write could start in mid-

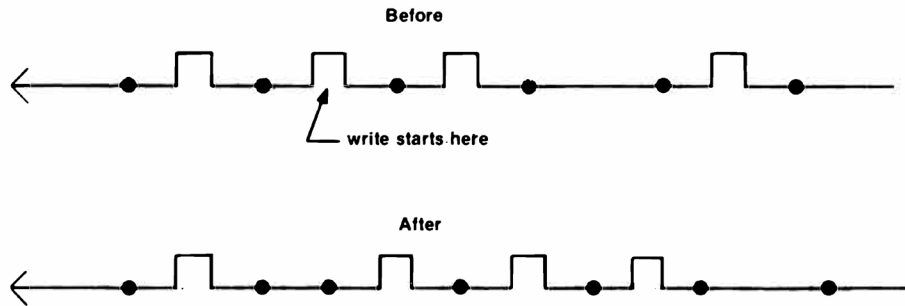


Figure 3.11 Writing Out of Sync

byte (see Figure 3.10). For this reason, the length of Gap 2 varies from five to ten bytes. This is not a problem as long as the difference in positioning is not too great. To insure the integrity of Gap 2 when writing a data field, five self-sync bytes are written prior to writing the Data Field itself. This serves two purposes. Since relatively little time is spent decoding an address field, the five bytes help place the Data Field near its original position. Secondly, and more importantly, the five self-sync bytes are the minimum number required to guarantee read-synchronization. It is probable that, in writing a Data Field, at least one sync byte will be destroyed. This is because, just as in reading bits on the track, the write may not begin on a byte boundary, thus altering an existing byte. Figure 3.11 illustrates this.

Gap 3 appears after each Data Field and before each Address Field. It is longer than Gap 2 and care is taken to make sure it ranges from 16 to 28 bytes in length. It is quite similar in purpose to Gap 2. Gap 3 allows the additional time needed to manipulate the data that has been read before the next sector is to be read. The length of Gap 3 is not as critical as that of Gap 2. If the following Address Field is missed, ProDOS can always wait for the next time it spins around under the read/write head (one revolution of the disk at most). Since Address Fields are never rewritten, there is no problem with Gap 3 providing synchronization, since only the first part of the gap can be overwritten or damaged (see Figure 3.10 for clarity).

ADDRESS FIELDS

An examination of the contents of the two types of field is in order. The Address Field contains the address or identifying information about the Data Field which follows it. The volume, track, and sector number of any given sector can be thought of as its “address,” much like a country, city, and street number might identify a house. As shown previously in Figure 3.6, there are

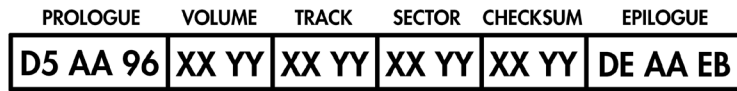


Figure 3.12 Address Field

a number of components which make up the Address Field. A more detailed illustration is given in Figure 3.12.

Each byte of the Address Field is encoded into two bytes when written to the disk. Appendix C describes the “4 and 4” method used for Address Field encoding.

The **prologue** consists of three bytes which form a unique sequence, found in no other component of the track. This fact enables ProDOS to locate an Address Field with almost no possibility of error. The three bytes are \$D5, \$AA, and \$96. The \$D5 and \$AA are reserved (never written as data), thus insuring the uniqueness of the prologue. The \$96, following this unique string, indicates that the data following constitutes an Address Field (as opposed to a Data Field). The address information follows next, consisting of the **volume***, **track**, and **sector** number and a checksum. This information is absolutely essential for ProDOS to know where it is positioned on a particular diskette. The **checksum** is computed by exclusive-ORing the first three pieces of information, and is used to verify its integrity. Lastly follows the **epilogue**, which contains the three bytes \$DE, \$AA, and \$EB. The \$EB is only partly written during initialization, and is therefore never verified when an Address Field is read. The epilogue bytes are sometimes referred to as **bit-slip marks**, which provide added assurance that the drive is still in sync with the bytes on the disk. These bytes are probably unnecessary, but do provide a means of double checking.

DATA FIELDS

The other field type is the Data Field. Much like the Address Field, it consists of a **prologue**, **data**, **checksum**, and an **epilogue** (refer to Figure 3.13). The Prologue differs only in the third byte. The bytes are \$D5, \$AA, and \$AD, which again form a unique sequence, enabling ProDOS to locate the beginning of the sector data. The data consists of 342 bytes of encoded data. (The encoding scheme used is quite complex and is documented in detail in Appendix C.) The data is followed by a checksum byte, used to verify the integrity of the data

* Volume number is a leftover from earlier operating systems and is not used by ProDOS.

just read. The epilogue portion of the Data Field is absolutely identical to the epilogue in the Address Field and serves the same function.

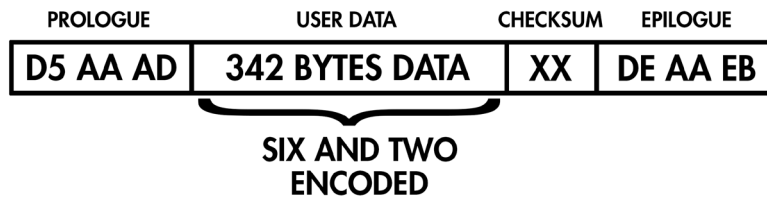


Figure 3.13 Data Field

DISK II BLOCK AND SECTOR INTERLEAVING

Because the Disk II is such an integral part of the Apple II family of machines, it is important that it perform efficiently. One major factor in disk drive performance is how the data is arranged on the diskette. Because the diskette spins and the head that reads and writes the data is stationary, it is necessary to wait for a particular portion of a given track to pass by. This waiting (rotational delay) can add significant time to a disk access if the data is poorly arranged. **Interleaving** (or skewing) is the arranging of data at the block or sector level to maximize access speed. It effectively places a gap between blocks or sectors that will normally be accessed sequentially, allowing sufficient time for internal housekeeping before the next one appears. In general, if blocks or sectors are poorly arranged on a track, it is usually necessary to wait an entire revolution of the diskette before the next desired block or sector can be accessed.

The first versions of Apple's operating system used **physical interleaving** on the disk. (That is, sectors were written in a particular order on the diskette.) A

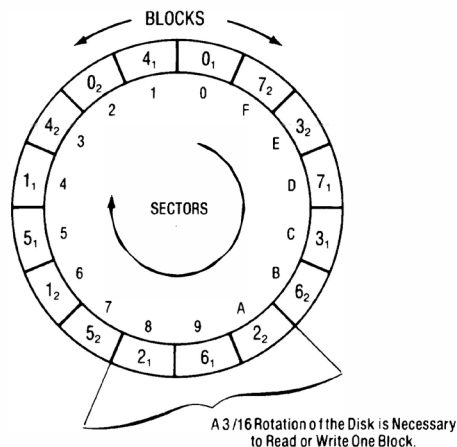


Figure 3.14 Block Interleaving (Track 0)

number of different schemes were used in an attempt to maximize performance. This worked reasonably well, but because different methods were used for different operations, performance suffered. Later versions standardized the physical interleaving (as sequential), and used a software method to try to maximize performance. An attempt was also made to standardize some operations, but performance still was not optimal as evidenced by a proliferation of “fast” DOSs.

ProDOS provides an impressive improvement over Apple’s earlier operating systems. Several factors account for the dramatic improvement. The routine to read data is significantly faster, minimizing the delay occurring between read operations. The data is dealt with in larger pieces (512 bytes vs. 256 bytes), lowering the number of requests to the code that actually reads and writes data (Device Driver). And almost all operations involve files stored on sequential blocks. As a disk begins to get full, this will not always be possible and some file will be discontinuous; but for the most part, all operations (loading ProDOS or Applesoft BASIC, reading or writing to files or a directory) involve data in contiguous pieces. This greatly simplifies the problem of finding an optimal interleaving for disk accesses.

In ProDOS, the interleaving is done in software. The 16 sectors are in numerically ascending order on the diskette (0, 1, 2, . . . 15), and are not physically interleaved at all. An algorithm is used to translate block numbers into physical sector numbers used by the ProDOS device driver. For example, if the block number requested were 2, this would be translated to track 0, physical sectors 8 and A*. Figure 3.14 illustrates the concept of **software interleaving** and Table 3.1 shows the mapping of physical sectors to blocks for a Disk II or compatible drive.

There are two kinds of interleaving to consider in the case of ProDOS. First, there is the interleaving of the two sectors that make up a block. This will be referred to as **intra-block** or “within block” interleaving. Second, there is the interleaving between blocks on a given track. This will be referred to as **inter-block** or “between block” interleaving. It should be noted that we are concerned primarily with delays within ProDOS and the Disk II Device Driver, and not with delays that may be present in various application packages.

* Those familiar with DOS 3.3 should note that physical sector numbers and DOS 3.3 sector numbers are not the same. Most “disk utilities” use DOS 3.3 sector numbers and not physical sector numbers. The bottom of Table 3.1 shows how DOS 3.3 sector numbers are related to ProDOS block numbers.

INTRA-BLOCK INTERLEAVING

When ProDOS accesses a block, it must of course access the two sectors that make up that block. There is a small delay after the device driver has accessed the first sector, before it can access the second sector. This delay is different for Read and Write operations. The Read operation is so fast that the disk can read two sectors in a row. However, the Write operation takes longer, so for optimal performance there must be a gap between the two sectors that make up a block. If there wasn't a gap, an entire revolution of the diskette would be required for each block written. A single sector provides a sufficient gap, so intra-block interleaving (within the block) consists of **one sector**. The result is that ProDOS is able to write to a given block as rapidly as is possible. Some time is lost when reading a block, but no other interleaving scheme would provide the same overall efficiency. Intra-block interleaving is illustrated in Figure 3.15.

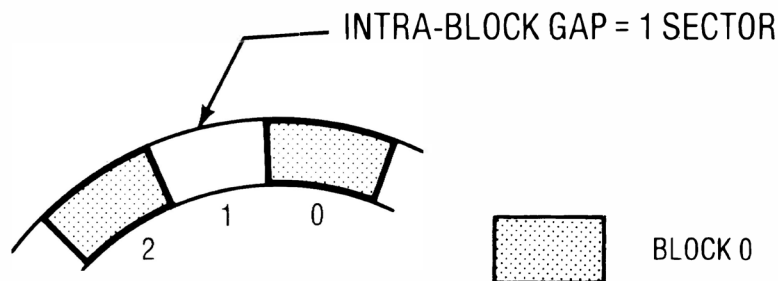


Figure 3.15 Intra-Block Interleaving (Within Block)

INTER-BLOCK INTERLEAVING

When ProDOS accesses a number of blocks as required in most disk operations (i.e. reading or writing a directory or a file), another kind of interleaving is involved. There will be a delay between accesses, but it is now between blocks rather than sectors. There is relatively little difference in delay time in the MLI itself between reading and writing—almost all the difference occurs in the device driver. However, when ProDOS writes a block that is already allocated (i.e. part of an existing directory or file), it always reads that block before writing to it. This requires an entire revolution of the diskette regardless of how the interleaving is done. It turns out that, just as for intra-block operations, a **single sector** is a sufficient gap for reading blocks. Inter-block interleaving is illustrated in Figure 3.16.

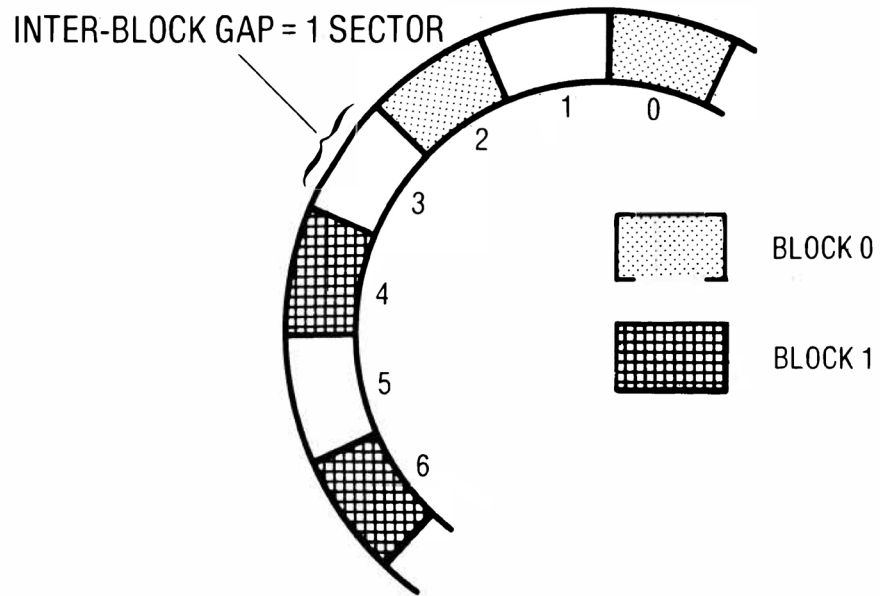


Figure 3.16 Inter-Block Interleaving (Between Block)

READING OR WRITING A BLOCK

Assume that we wish to access block 2. ProDOS passes the request to the device driver which in turn converts the block number into its track and sector representation (see Figure 3.14). The arm is moved to the proper track (0) and then a sector is read. This could be any sector, because the diskette is spinning. Sectors are continually read until sector 8 is found. The following two sectors are then read (9 and A) which completes the read of block 2 (sectors 8 and A). Depending on where we start on the track, we could read between 3 and 18 sectors. The same process occurs when writing a single block, with one small difference. After sector 8 is located and written to, the delay required to ready the data for sector A will cause us to miss reading sector 9. This does not alter the amount of rotation necessary to complete the task. To summarize, the time required to either read or write a single block consists of two factors. (We are assuming the track has already been located). First, there is the time required to **locate** the first sector of the block—this is variable and ranges between 0 and the time of one full rotation of the diskette. Second is the time required to actually **read or write** the two sectors that make up the block—this is fixed and always requires 3/16 rotation of the diskette.

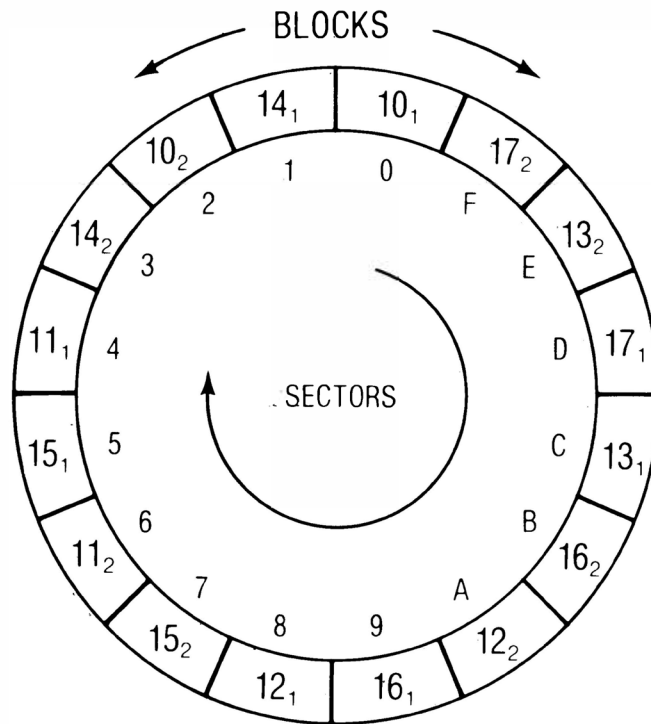


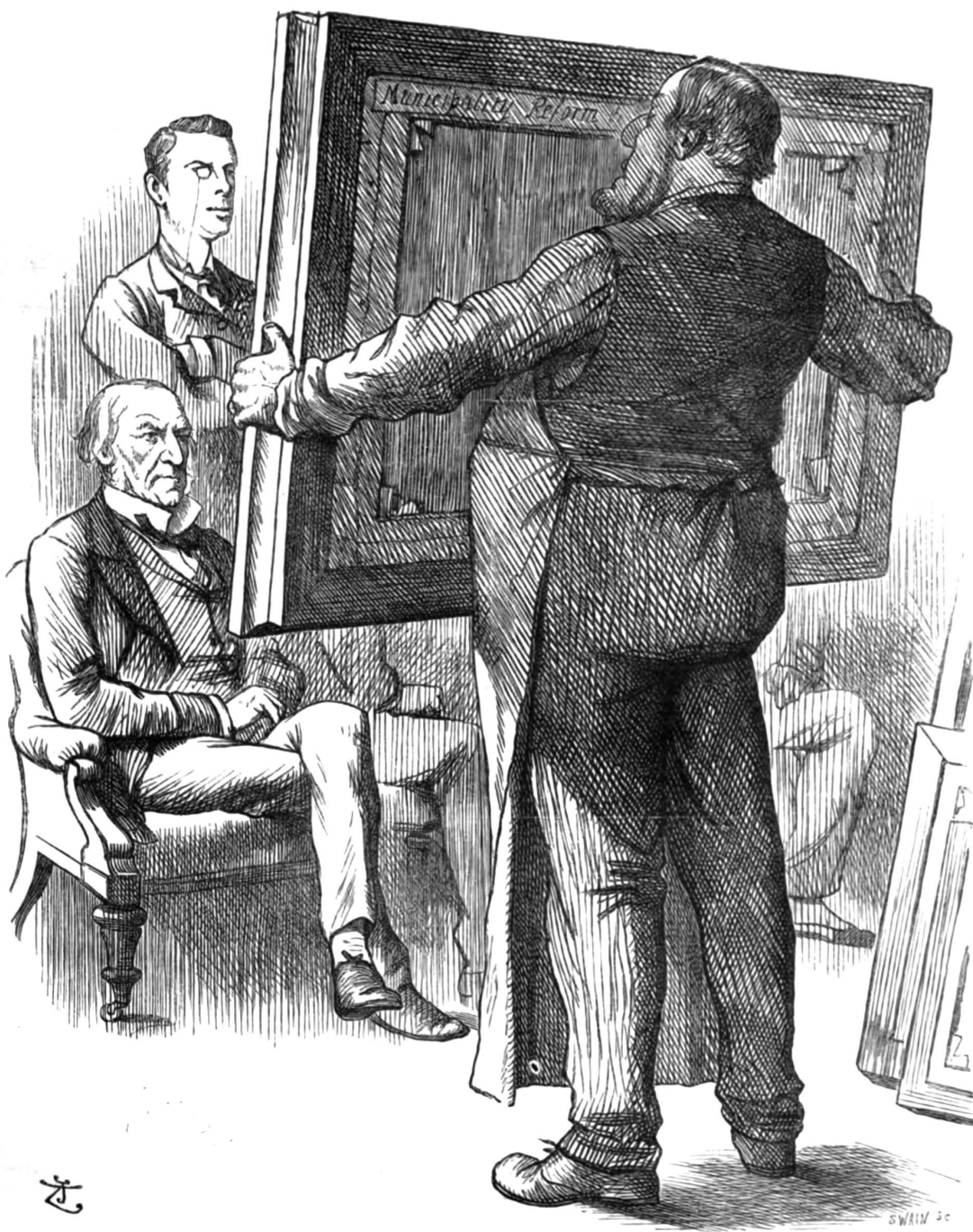
Figure 3.17 Example of Block Interleaving of Track 2

READING OR WRITING CONSECUTIVE BLOCKS

Let's examine what occurs when a number of blocks are accessed during reading or writing of a typical file. We will assume the file is reasonably large and takes up a number of blocks. We will confine our observation to a single track in which eight blocks comprise the file of interest. We will assume track 2, which contains blocks 10 through 17 (as in Figure 3.17), and we will further assume that the blocks will be accessed sequentially. When the read/write head moves to track 2, we will start reading sectors until the appropriate sector is found (0 in this case). Then each sector is read until all eight blocks are found. This will require exactly **two revolutions** of the disk. Writing takes significantly longer because each block is read before being written to. Therefore, once the first sector of the block in question is located, one entire revolution is necessary to write each block. Upon writing a block, ProDOS is able to locate the next block immediately, read it, wait one revolution, and write it. A total of **ten revolutions** is required to write an entire track as opposed to two revolutions to read it.

Table 3.1 ProDOS Block Conversion Table for Diskettes

	PHYSICAL SECTOR							
	0&2	4&6	8&A	C&E	1&3	5&7	9&B	D&F
TRACK 00	000	001	002	003	004	005	006	007
TRACK 01	008	009	00A	00B	00C	00D	00E	00F
TRACK 02	010	011	012	013	014	015	016	017
TRACK 03	018	019	01A	01B	01C	01D	01E	01F
TRACK 04	020	021	022	023	024	025	026	027
TRACK 05	028	029	02A	02B	02C	02D	02E	02F
TRACK 06	030	031	032	033	034	035	036	037
TRACK 07	038	039	03A	03B	03C	03D	03E	03F
TRACK 08	040	041	042	043	044	045	046	047
TRACK 09	048	049	04A	04B	04C	04D	04E	04F
TRACK 0A	050	051	052	053	054	055	056	057
TRACK 0B	058	059	05A	05B	05C	05D	05E	05F
TRACK 0C	060	061	062	063	064	065	066	067
TRACK 0D	068	069	06A	06B	06C	06D	06E	06F
TRACK 0E	070	071	072	073	074	075	076	077
TRACK 0F	078	079	07A	07B	07C	07D	07E	07F
TRACK 10	080	081	082	083	084	085	086	087
TRACK 11	088	089	08A	08B	08C	08D	08E	08F
TRACK 12	090	091	092	093	094	095	096	097
TRACK 13	098	099	09A	09B	09C	09D	09E	09F
TRACK 14	0A0	0A1	0A2	0A3	0A4	0A5	0A6	0A7
TRACK 15	0A8	0A9	0AA	0AB	0AC	0AD	0AE	0AF
TRACK 16	0B0	0B1	0B2	0B3	0B4	0B5	0B6	0B7
TRACK 17	0B8	0B9	0BA	0BB	0BC	0BD	0BE	0BF
TRACK 18	0C0	0C1	0C2	0C3	0C4	0C5	0C6	0C7
TRACK 19	0C8	0C9	0CA	0CB	0CC	0CD	0CE	0CF
TRACK 1A	0D0	0D1	0D2	0D3	0D4	0D5	0D6	0D7
TRACK 1B	0D8	0D9	0DA	0DB	0DC	0DD	0DE	0DF
TRACK 1C	0E0	0E1	0E2	0E3	0E4	0E5	0E6	0E7
TRACK 1D	0E8	0E9	0EA	0EB	0EC	0ED	0EE	0EF
TRACK 1E	0F0	0F1	0F2	0F3	0F4	0F5	0F6	0F7
TRACK 1F	0F8	0F9	0FA	0FB	0FC	0FD	0FE	0FF
TRACK 20	100	101	102	103	104	105	106	107
TRACK 21	108	109	10A	10B	10C	10D	10E	10F
TRACK 22	110	111	112	113	114	115	116	117



HGR to the MAX

Chapter 4

As was described in Chapter 3, a 16-sector diskette consists of 560 data areas of 256 bytes each, called sectors. These sectors are arranged on the diskette in 35 concentric rings, called tracks, of 16 sectors each. The way ProDOS allocates these tracks of sectors is the subject of this chapter.

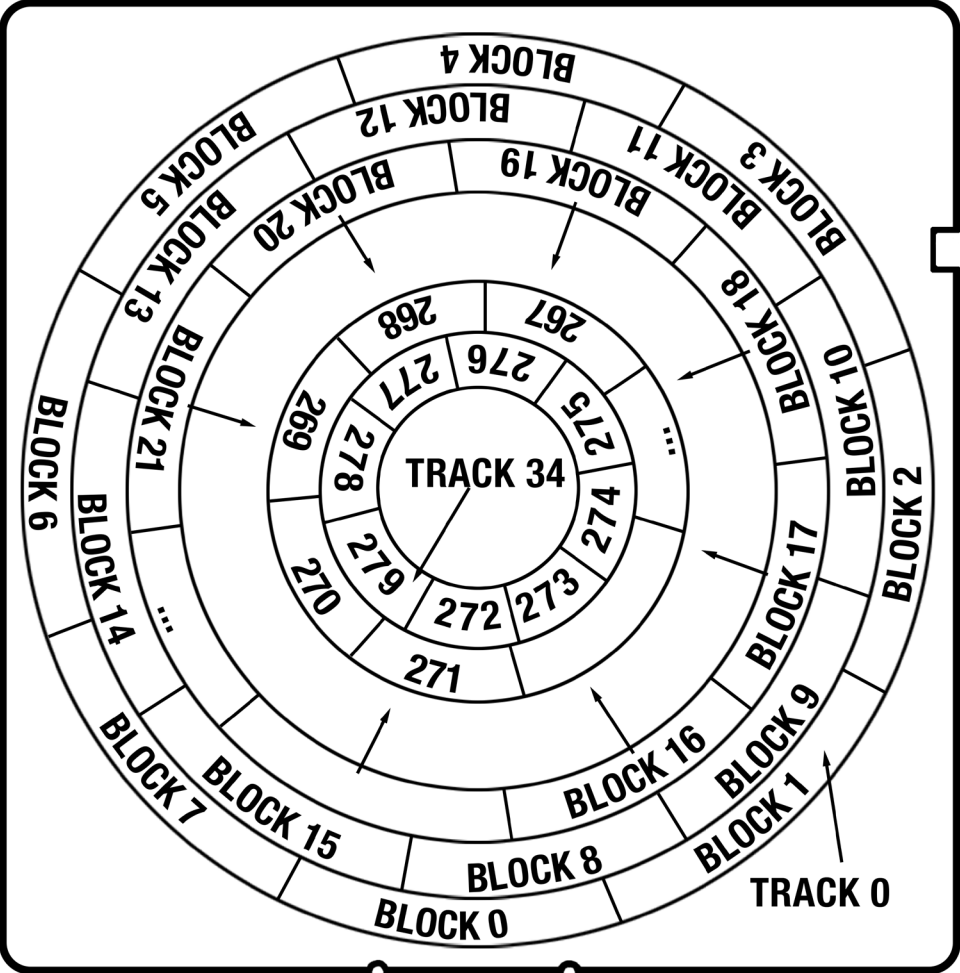


Figure 4.1 Blocks on a Diskette

THE DISKETTE VOLUME

ProDOS defines a **volume** to be any (usually direct access) individual mass storage media. The discussion which follows assumes this media to be a single 35-track diskette, but all of the structures presented here are identical for other diskette sizes and even for a hard disk such as the Apple ProFile. Another interesting point is that the structure of a ProDOS volume is almost identical to that of an Apple III SOS volume. This fact allows greater data compatibility between the two operating systems.

To make the allocation of sectors more manageable, ProDOS pairs them up to form 512-byte **blocks**. Since there are 16 sectors per track and 560 sectors per diskette volume, there are eight blocks per track and 280 blocks per volume. These blocks are numbered from 0 to 279 (decimal) or \$0000 to \$0117 (hexadecimal). The arrangement of blocks on a diskette is shown in Figure 4.1. Of course, on a real diskette, skewing (discussed in Chapter 3) would reorder the blocks on any given track, but, for the purposes of this discussion, the blocks can be assumed to be stored sequentially.

A file, be it **BAS**, **BIN**, **TXT**, or **SYS** type, consists of one or more blocks containing data. Since a block is the smallest unit of allocatable space on a ProDOS volume, a file will use up at least one block even if it is less than 512 bytes long; the remainder of the block is wasted. Thus, a file containing 600 characters (or bytes) of data will occupy one entire block and 88 bytes of another with 424 bytes wasted. Knowing that there are 280 blocks on a diskette, one might expect to be able to use up to 280 times 512 or 143,360 bytes of space on a diskette for files. Actually, the largest file that can be stored in 271

/EXAMPLE

NAME	TYPE	BLOCKS	MODIFIED	CREATED	ENDFILE	SUBTYPE
BASFILE	BAS	1	<NO DATE>	<NO DATE>	109	
TXTFILE	TXT	1	<NO DATE>	<NO DATE>	9	R= 64
BINFILE	BIN	1	<NO DATE>	<NO DATE>	48	A=\$03D0

BLOCKS FREE: 270 BLOCKS USED: 10 TOTAL BLOCKS: 280

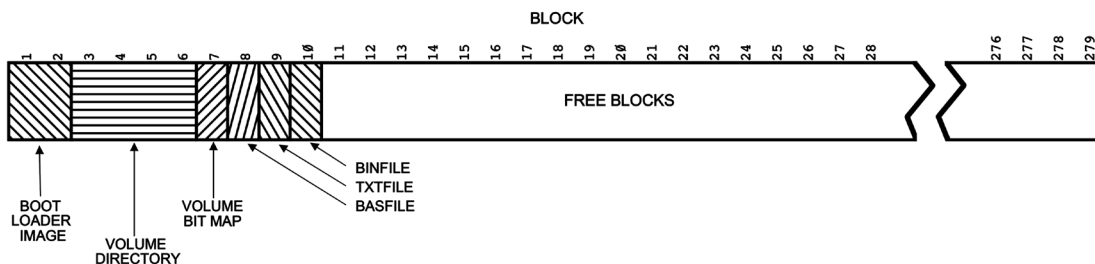


Figure 4.2 Block Usage on an Example Diskette

blocks long (or 138,752 bytes). The reason for this is that some of the blocks on the diskette volume must be used for what is called **overhead**.

VOLUME OVERHEAD

Overhead blocks contain the image of the ProDOS bootstrap loader (which is loaded by the ROM on your diskette controller card and, in turn, loads the ProDOS system files into memory), a list of file names and locations of the files on the diskette, and an accounting of the blocks which are free for use by new files or for expansions of existing ones. An example of the way ProDOS uses blocks is given in Figure 4.2.

Notice that in the case of this diskette volume, system overhead (that part of the diskette which does not actually contain files) falls entirely on track 0 of the diskette (blocks 0 through 7). In fact, there is room for one block's worth of file data on track 0 (block 7). The first block (block 0) is always devoted to the image of the bootstrap loader. (Block 1 is the SOS bootstrap loader.) Following these, and always starting at block 2, is the **Volume Directory**. The

Volume Directory is the “anchor” of the entire volume. On any diskette (or hard disk for that matter) for any version of ProDOS, the first or “key” block of the Volume Directory is always in the same place—block 2. Since files can end up anywhere on the diskette, it is through the Volume Directory key block that ProDOS is able to find them. Thus, just as the card catalog is used to locate a book in a library, the Volume Directory is the master index to all of the files on a volume. In addition to describing the name, attributes, and placement of

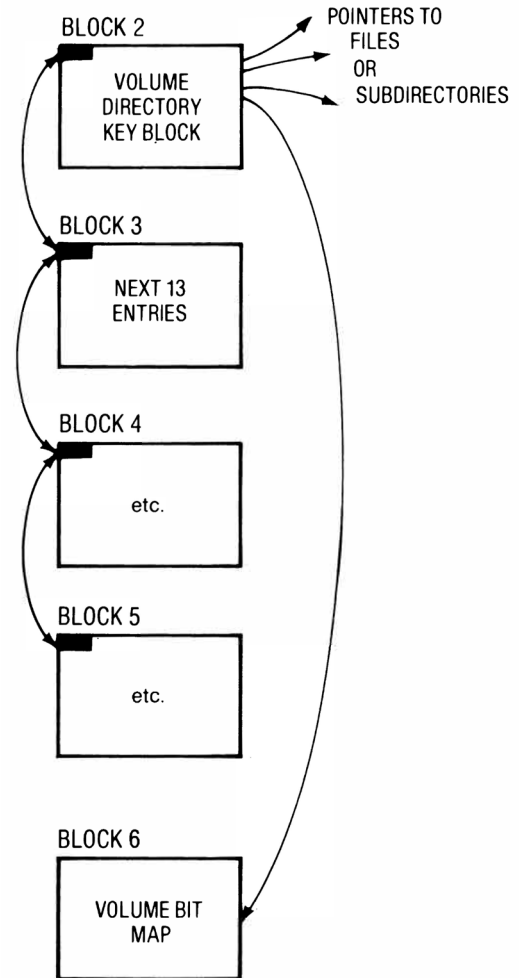


Figure 4.3 Linking of Volume Directory and Volume Bit Map

each file, it also contains the block number of the **Volume Bit Map** which will be described next. The first four bytes of every Volume Directory block are reserved for “pointers” to (the block numbers of) the previous Volume Directory block and the next Volume Directory block. This structure is called a doubly-linked list and is handy in that, from any block, it is easy to move forward or backward through the directory entries. The Volume Directory and Volume Bit Map are diagrammed in Figure 4.3.

VOLUME SPACE ALLOCATION – THE VOLUME BIT MAP

When a diskette volume is first formatted, only the first seven blocks described above are marked in use. All of the remainder of the diskette blocks are considered “free” for use with files yet to be created. Each time a new block is required for a file, the free block with the lowest number is used. To keep track of which blocks have been used and which are free, ProDOS maintains one block as the Volume Bit Map. The Volume Bit Map is located by following a pointer in the Volume Directory, however, it is almost always in block 6. It consists of 512 bytes, each byte representing eight blocks on the volume. If the bytes are examined in binary form, each consists of eight bits having a value of one or zero. Thus, if block zero is in use as it always is, then the first byte’s first bit is set to zero. If the ninth block (block 8) is free, then the first bit of the second byte is set to one. Since there are many more bits in the Volume Bit Map (4096 in all) than there could ever be blocks on a diskette, only the first 280 (or 35 bytes) are used. For a 5 megabyte hard disk, like the Apple ProFile, 1241 bytes are needed; in this case, since the number of blocks on the volume

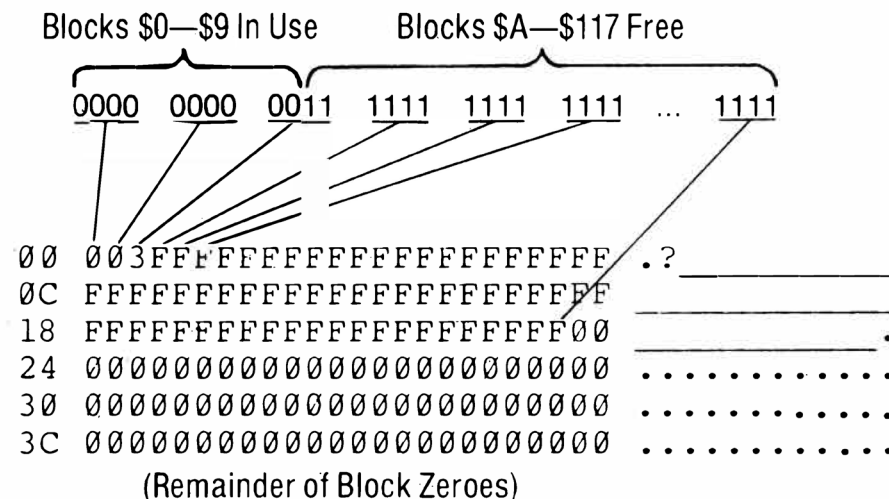


Figure 4.4 Example Volume Bit Map

is stored in the Volume Directory, ProDOS automatically knows to expect a bigger Volume Bit Map—one which is three blocks long. Bits which do not correspond to a real block (because it would be past the end of the volume) are set to zero. An example of a Volume Bit Map for the volume mapped in Figure 4.2 is given in Figure 4.4. Notice that, since three 1-block files have been allocated, a total of ten blocks are marked “in use.”

THE VOLUME DIRECTORY

When ProDOS must find a specified file by name, it first reads block 2 of the diskette, the **key block** of the Volume Directory. If the filename is not found in this block, the next directory block is read, following the pointer in the third and fourth bytes of the current block. Typically, the Volume Directory blocks occupy blocks 2 through 5 of a volume. Of course, as long as a block pointer exists, linking one block to the next, and the first Volume Directory block is block 2, ProDOS does not really care where the rest of the directory blocks are located. Figure 4.5

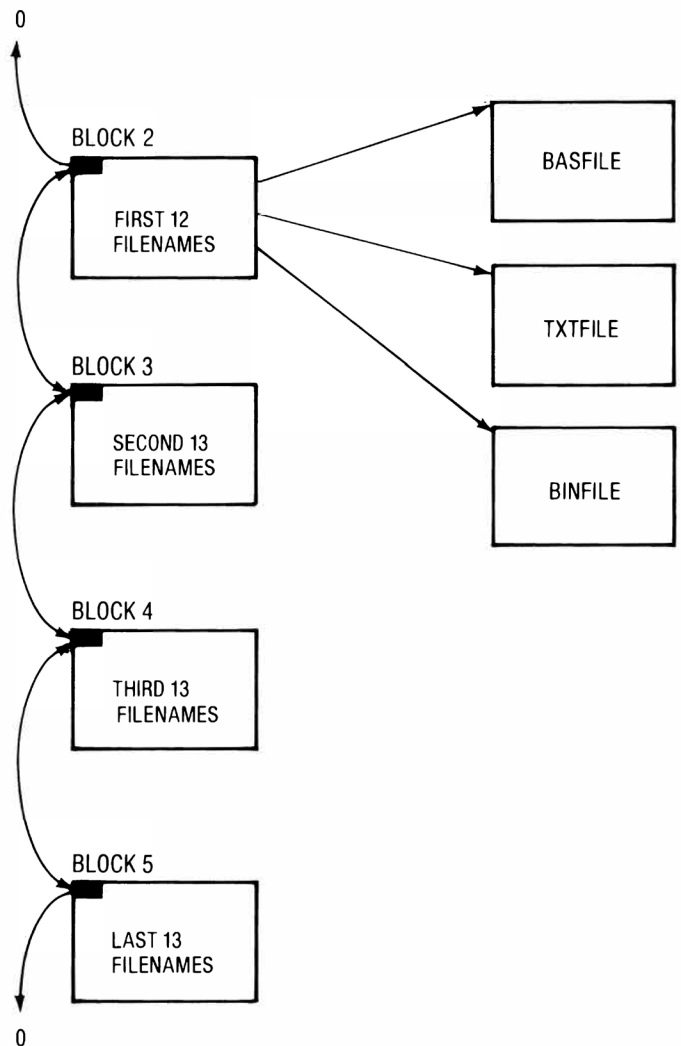


Figure 4.5 The Volume Directory

diagrams the Volume Directory for the example given in Figure 4.2. The figure shows the “next block” pointer (bytes +2 and +3 in the block) of block

2 in the Volume Directory, as an arrow pointing to block 3. Each block, in turn, has block numbers in the same relative location (+0, +1 and +2, +3) which point backward to the previous block and forward to the next block respectively. If no previous or next block exists, a block number of zero is used to indicate this (block 0, being part of the boot image, would never be a valid block number for a directory or file block, so this is a safe convention). The first block in the Volume Directory (the key block) contains a special entry called the **header** which describes the directory itself and the characteristics of the volume, etc. This is followed by 12 file descriptive entries. All Volume Directory blocks other than the key block contain descriptions of up to 13 files each. (In practice, these entries can also be used to describe subdirectories, but this will be covered in detail later in the chapter.) Thus, with four Volume Directory blocks, a total of 4 times 13 less 1 (for the Volume Directory Header entry) or 51 files may be described.

THE VOLUME DIRECTORY HEADER

The Volume Directory Header is the first entry in the first block of the Volume Directory. As such, its first byte follows the four bytes of next/previous block pointers, so its first byte is at +\$04. A description of its format follows:*

BLOCK BYTE	DESCRIPTION
\$04	STORAGE_TYPE/NAME_LENGTH: The first nibble (top four bits) of this byte describes the type of entry. In this case, this is a Volume Directory Header so this nibble is \$F. The low four bits are the length of the name in the next field (the volume name).
\$05-\$13	VOLUME_NAME: A 15-byte field containing the name of this column. The actual length is defined by NAME_LENGTH above; the remainder of the field is ignored. No “/” is present as the first character since this is only used to delimit different level names but is not part of the names themselves.
\$14-\$1B	Reserved for future use. Usually zeroes.
\$1C-\$1F	CREATION: The date and time of creation (formatting) of this volume. This field is zero if no date was assigned. The format of the field is as follows: BYTE 0 and 1 — yyyyyyym mmmddddd year/month/day BYTE 2 and 3 — 000hhhhh 00mmmmmm hours/minutes where each letter above represents one binary bit. This is the standard form for all create and modify date/time stamps in directories.

* Unless otherwise indicated, all multiple byte numeric values, such as block numbers, EOF marks, etc., are stored least significant byte first, most significant byte last (LO/HI).

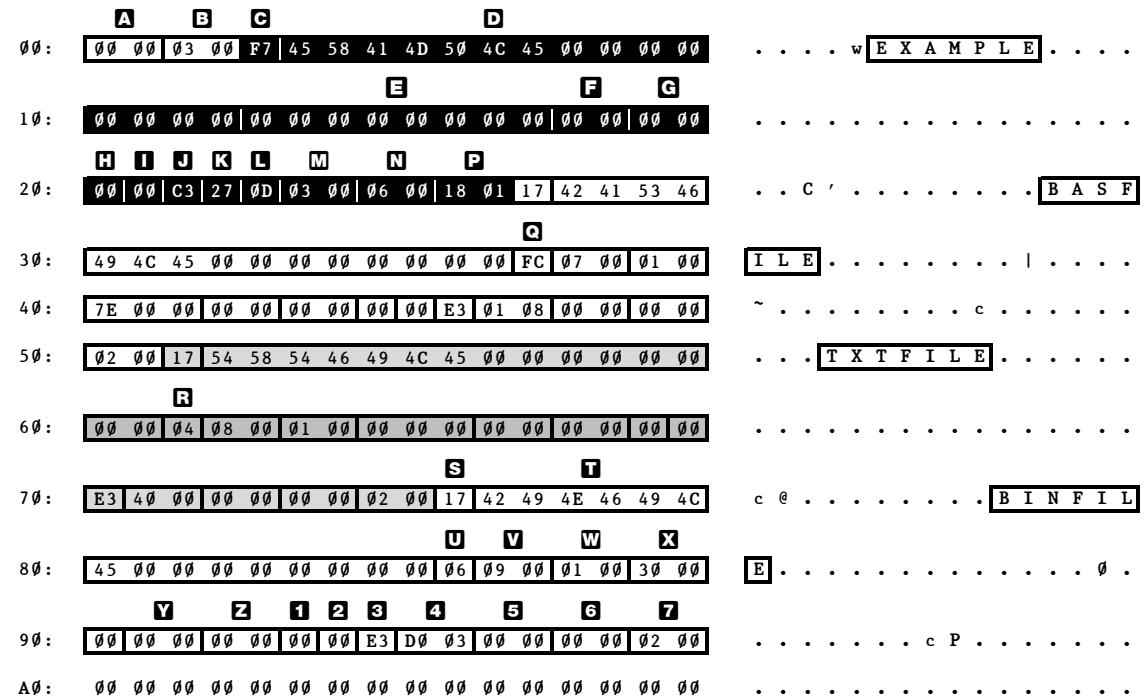
BLOCK BYTE	DESCRIPTION
\$20	VERSION: The ProDOS version number under which this volume was formatted. This field tells later versions of ProDOS not to expect to find any fields which were defined by Apple after this version of ProDOS was released. This field indicates the level of upward compatibility between versions. Under ProDOS 1.0, its value is zero.
\$21	MIN_VERSION: Minimum version of ProDOS which can access this volume. A value in this field implies that significant changes were made to the field definitions since prior versions of ProDOS were in use and these older versions would not be able to successfully interpret the file structure of this volume. This field indicates the level of downward compatibility between versions. Under ProDOS 1.0, its value is zero.
\$22	ACCESS: The bits in the flag byte define how the directory may be accessed. The bit assignments are as follows: \$80 — Volume may be destroyed (reformatted) \$40 — Volume may be renamed \$20 — Volume directory has changed since last backup \$02 — Volume directory may be written to \$01 — Volume directory may be read All other bits are reserved for future use.
\$23	ENTRY_LENGTH: Length of each entry in the Volume Directory in bytes (usually \$27).
\$24	ENTRIES_PER_BLOCK: Number of entries in each block of the Volume Directory (usually \$0D). Note that the Volume Directory Header is considered to be an entry.
\$25-\$26	FILE_COUNT: Number of active entries in the Volume Directory. An active entry is one which describes a file or subdirectory which has not been deleted. This count does not include the Volume Directory Header. Note that this field's name is a bit misleading since the count also includes subdirectory entries.
\$27-\$28	BIT_MAP_POINTER: The block number of the first block of the Volume Bit Map described earlier. This value is usually 6.
\$29-\$2A	TOTAL_BLOCKS: The total number of blocks on this volume. \$0118 is for a 35-track diskette (280 decimal). This number may be used to compute the number of blocks in the Volume Bit Map as described earlier.

FILE DESCRIPTIVE ENTRIES

Each file (or subdirectory) on a volume has a File Descriptive Entry in the Volume Directory or another directory. These entries all have the same format:

BYTE OFFSET	DESCRIPTION
\$00	<p>STORAGE_TYPE/NAME_LENGTH: The first nibble (top four bits) of this byte describes the type of entry. Currently assigned values are:</p> <p>\$0 - Deleted entry. Available for reuse</p> <p>\$1 - File is a seedling (only one data block)</p> <p>\$2 - File is a sapling (2 to 256 data blocks)</p> <p>\$3 - File is a tree (257 to 32768 data blocks)</p> <p>\$D - File is subdirectory</p> <p>\$E - Reserved for Subdirectory Header entry</p> <p>\$F - Reserved for Volume Directory Header entry</p> <p>The low four bits are the length of the file or subdirectory name in the next field. When a file is deleted, a \$00 is stored in this byte.</p>
\$01-\$0F	<p>FILE_NAME: a 15-byte field containing the name of this file. The actual length is defined by NAME_LENGTH above; the remainder of the field is ignored.</p>
\$10	<p>FILE_TYPE: Primary file type. The hexadecimal value of this byte gives the file type as show in the following table:</p>

TYPE	NAME	DESCRIPTION
\$00		Typeless file
\$01	BAD	Bad block(s) file
\$04	TXT	Text file (ASCII text, msb off)
\$06	BIN	Binary file (8-bit binary image)
\$0F	DIR	Directory file
\$19	ADB	AppleWorks data base file
\$1A	AWP	AppleWorks word processing file
\$1B	ASP	AppleWorks spreadsheet file
\$EF	PAS	ProDOS PASCAL file
\$F0	CMD	ProDOS added command file
\$F1-\$F8		User defined file types 1 through 8
\$FC	BAS	Applesoft BASIC program file
\$FD	VAR	Applesoft stored variables file



POINTER FIELDS	OTHER FILES IN DIRECTORY
A (\$00) no previous DIR block	Q BAS file type
B (\$00) next DIR block is 3	R TXT file type
VOLUME DIRECTORY HEADER	S \$1 indicates seedling file \$7 characters in name
C \$F indicates Volume Directory Header volume name is 7 characters long	T filename is "BINFILE"
D volume name is "Example"	U filetype is \$06 (BIN)
E reserved	V pointer to key block of file
F creation date (\$0000 = Not Set)	W total blocks used by file
G creation time (\$0000 = Not Set)	X length of file (3 byte value)
H (\$00 = 1.0) version of ProDOS	Y creation date
I (\$00) minimum version of ProDOS	Z creation time
J (\$C3) access byte	1 version of ProDOS
K (\$27) entry length	2 minimum version of ProDOS
L (\$0D) entries per block	3 access byte
M (\$0003) file count	4 auxiliary type
N bit map pointer (Block \$0006)	5 last modification date
P total blocks	6 last modification time
	7 pointer to directory header block

Figure 4.6 Example Volume Directory Block

TYPE	NAME	DESCRIPTION
\$FE	REL	Relocatable object module file (EDASM)
\$FF	SYS	ProDOS system file

All other types are either SOS file types or are reserved by Apple for future use. See Appendix E for a complete list.

- \$11–\$12** **KEY_POINTER:** The block number of the key block of the file. In the case of a seedling file, this is the block number of the index block. For tree files, this is the block number of the master index block. (More on these file structures later.) If the file is a subdirectory, this is the block number of its first block.
- \$13–\$14** **BLOCKS_USED:** The total number of blocks used by this file including index blocks and data blocks. If the file is a subdirectory, this is the number of directory blocks.
- \$15–\$17** **EOF:** The location of the end of the file (**EOF**) as a 3-byte offset from the first byte. This can also be thought of as the length in bytes of a sequential file.
- \$18–\$1B** **CREATION:** The date and time of the creation of this file. This field is zero if no date was assigned. The format of the field is as follows:
 Byte 0 and 1 - **yyyyyyym mmmddddd** year/month/day
 Byte 2 and 3 - **000hhhhh 00mmmmmm** hours/minutes
 where each letter above represents one binary bit. This is the standard form for all create and modify date/time stamps in directories.
- \$1C** **VERSION:** The ProDOS version number under which this file was created. This field tells later versions of ProDOS not to expect to find any fields which were defined by Apple after this version of ProDOS was released. This field indicates the level of upward compatibility between versions. Under ProDOS 1.0, its value is zero.
- \$1D** **MIN_VERSION:** Minimum version of ProDOS which can access this file. A value in this field implies that significant changes were made to the file structure definition since prior versions of ProDOS were in use, and these older versions would not be able to successfully interpret the file structure of this file. This field indicates the level of downward compatibility between versions. Under ProDOS 1.0, its value is zero.
- \$1E** **ACCESS:** The bits in this flag byte define how the file may be accessed. The bit assignments are as follows:
\$80 - File may be destroyed
\$40 - File may be renamed
\$20 - File has changed since last backup

\$02 - File may be written to

\$01 - File may be read

All other bits are reserved for future use. An unlocked file's **ACCESS** is usually \$E3. If a file is locked, **ACCESS** will be set to \$21. Subdirectory files which have a non-zero **FILE_COUNT** field will be locked until all files described by them are deleted.

Value	Read?	Write?	Rename?	Delete?	ProDOS Use
\$20	-	-	-	-	
\$21	Y	-	-	-	Locked
\$22	-	Y	-	-	
\$23	Y	Y	-	-	
\$60	-	-	Y	-	
\$61	Y	-	Y	-	
\$62	-	Y	Y	-	
\$63	Y	Y	Y	-	
\$A0	-	-	-	Y	
\$A1	Y	-	-	Y	
\$A2	-	Y	-	Y	
\$A3	Y	Y	-	Y	
\$E0	-	-	Y	Y	
\$E1	Y	-	Y	Y	
\$E2	-	Y	Y	Y	
\$E3	Y	Y	Y	Y	Unlocked

\$1F-\$20 **AUX_TYPE**: Auxiliary type field whose contents depend upon **FILE_TYPE**. Common uses are as follows:

Type	Use
TXT	Random access record length (L from OPEN)
BIN	Load address for binary image (A from BSAVE)
BAS	Load address for program image (when SAVED)
VAR	Address of compressed variables images (when STORED)
SYS	Load address for system program (usually \$20000)

\$21-\$24 **LAST_MOD**: Date and time at which file was last modified. This field is zero if no date was assigned. Format is identical to **CREATION** above.

\$25-\$26 **HEADER_POINTER**: Block number of the key block for the directory which describes this file.

Figure 4.6 is an example of a typical Volume Directory block for the example introduced with Figure 4.2. In this case, there are only three files on the diskette so only the first three directory entries are filled in. The remaining directory entries have never been used and contain zeroes.

FILE STRUCTURES

One of ProDOS's major jobs is to keep track of the blocks which make up a file. When programming, the user need never know that a file is actually made up of one or more blocks scattered far and wide all over the diskette volume. ProDOS must make the file appear to the programmer to be a continuous stream of sequential data.

So far the files shown in the examples here have only had one block. This was done to avoid complicating the discussion of the Volume Directory. In practice, however, very few files are 512 bytes or less in length. ProDOS defines three file structures to handle files of different sizes:

- The Seedling — for files of 512 bytes or less
- The Sapling — for files with more than 512 bytes but less than 128K bytes of data
- The Tree — for files with more than 128K bytes of data up to 16 megabytes (16,777,216 bytes)

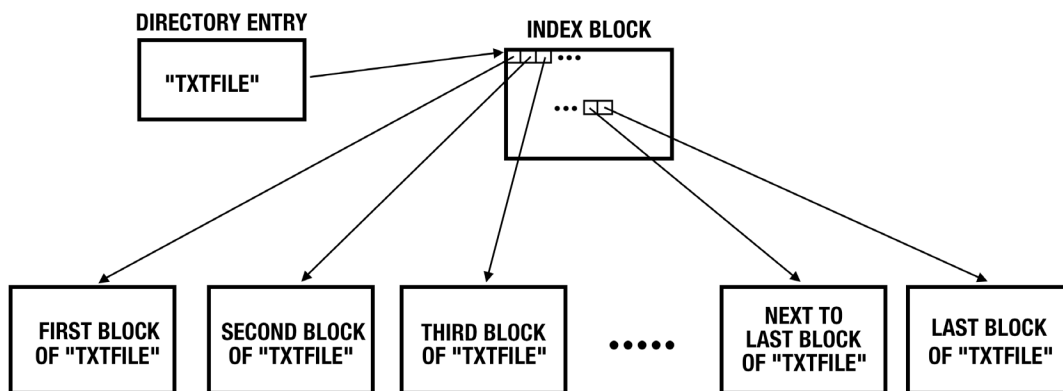


Figure 4.7 Sapling File Organization

Examples of **seedling** files have already been shown. A seedling file consists of a single **data block** whose number is stored in the **KEY_POINTER** field in the file entry of the directory. Thus, a seedling file, by definition, costs only one block of storage (and a file descriptive entry).

FIRST HALF OF INDEX BLOCK (LSBs)	A																B			
	00	08	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	00	00	00	00	00	00
	10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

SECOND HALF OF INDEX BLOCK (MSBs)	A																B			
	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

A \$0008 First data block of file is block \$8

B \$0016 Last data block of file is \$16

Figure 4.8 Example Sapling Index Block

For the purposes of this discussion, let us assume that we had run the following Applesoft BASIC program against our example disk volume from Figure 4-2.

```
10 PRINT CHR$(4);"OPEN TXTFILE,L64"
20 FOR I=0 TO 2
```

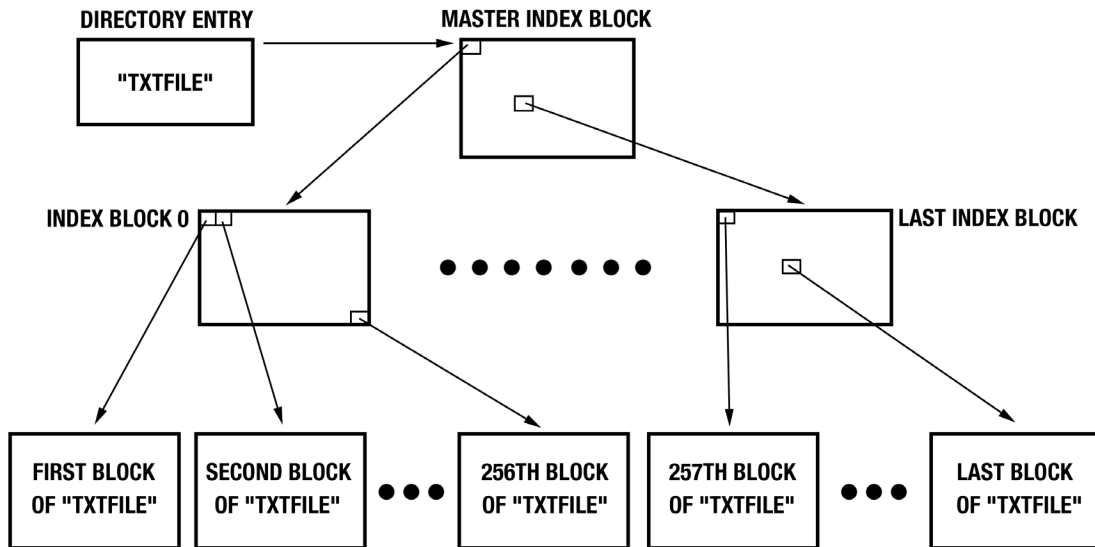


Figure 4.9 Tree File Organization

```

30 PRINT CHR$(4); "WRITE TXTFILE,R"; I
40 PRINT "RECORD"; I
50 NEXT I
60 PRINT CHR$(4); "CLOSE TXTFILE"
70 END

```

This program creates the **TEXT** file, "TXTFILE", with a record length of 64 bytes. It then writes three records containing the strings "RECORD0", "RECORD1", and "RECORD2". The total size of this file is then 3 times 64 or 192 bytes. Since this is less than 512 bytes, the file is stored as a seedling.

Now, assume that statement 20 is changed to read:

```
20 FOR I=0 TO 100
```

and the program is rerun. The file it creates will now contain 101 records of 64 bytes each, so the total size is 6464 bytes. As the ninth record is written (RECORD8), ProDOS discovers that the original seedling block is full. There is no room in the directory to store another block number, so ProDOS creates what is called an **index block**. This block contains the block numbers of each data block in the file in the order that they should be accessed. Using an index block, ProDOS can describe the file in a sequential and orderly way, even though its data blocks may not be physically contiguous (next to one another on the diskette). For example, if the previous data block in a file was 47, it is not necessary to store the data which follows it in block 48. Instead, any free

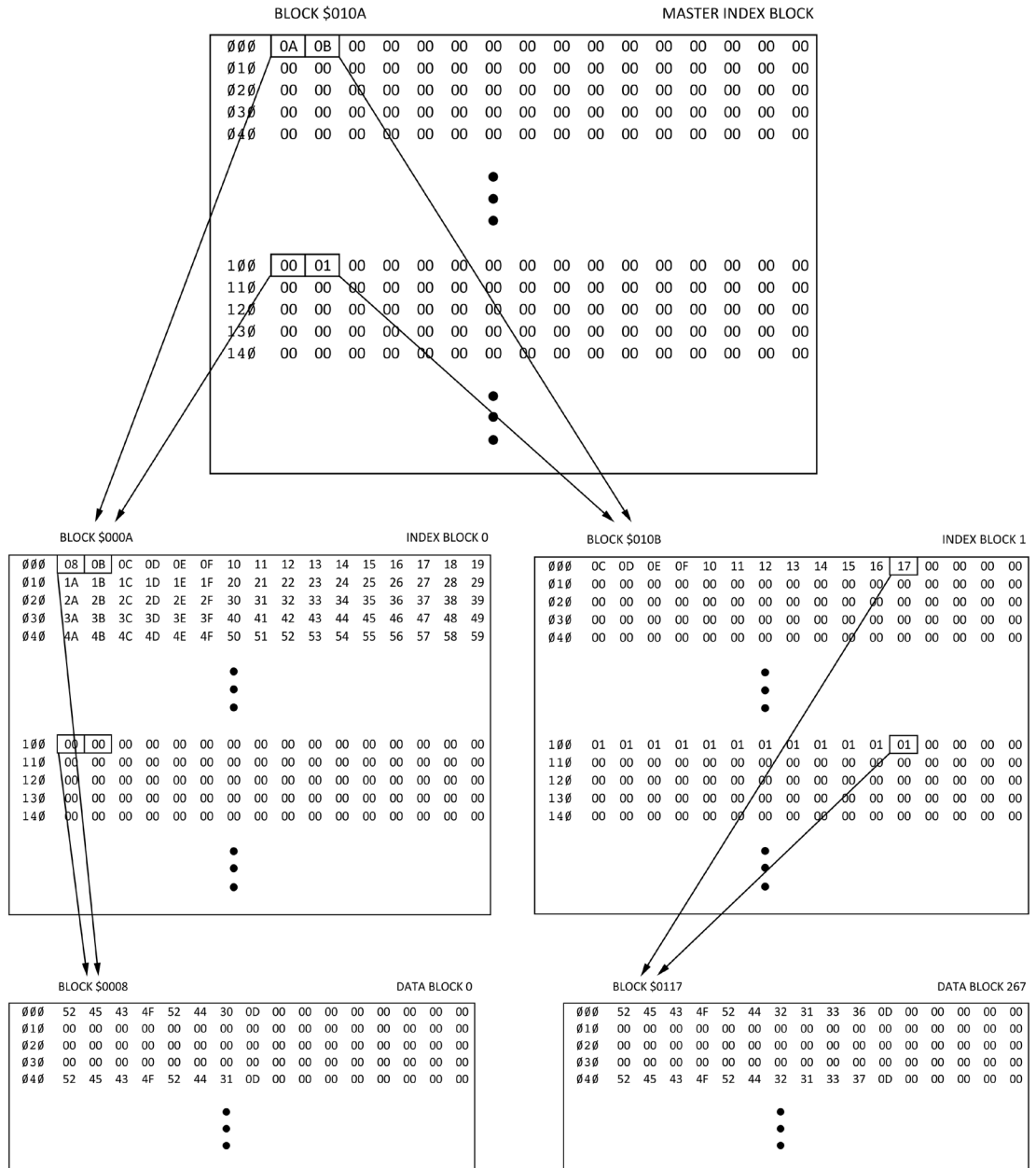


Figure 4.10 Example Tree File

block located anywhere on the diskette may be used simply by placing its block number next to 47's in the index block.

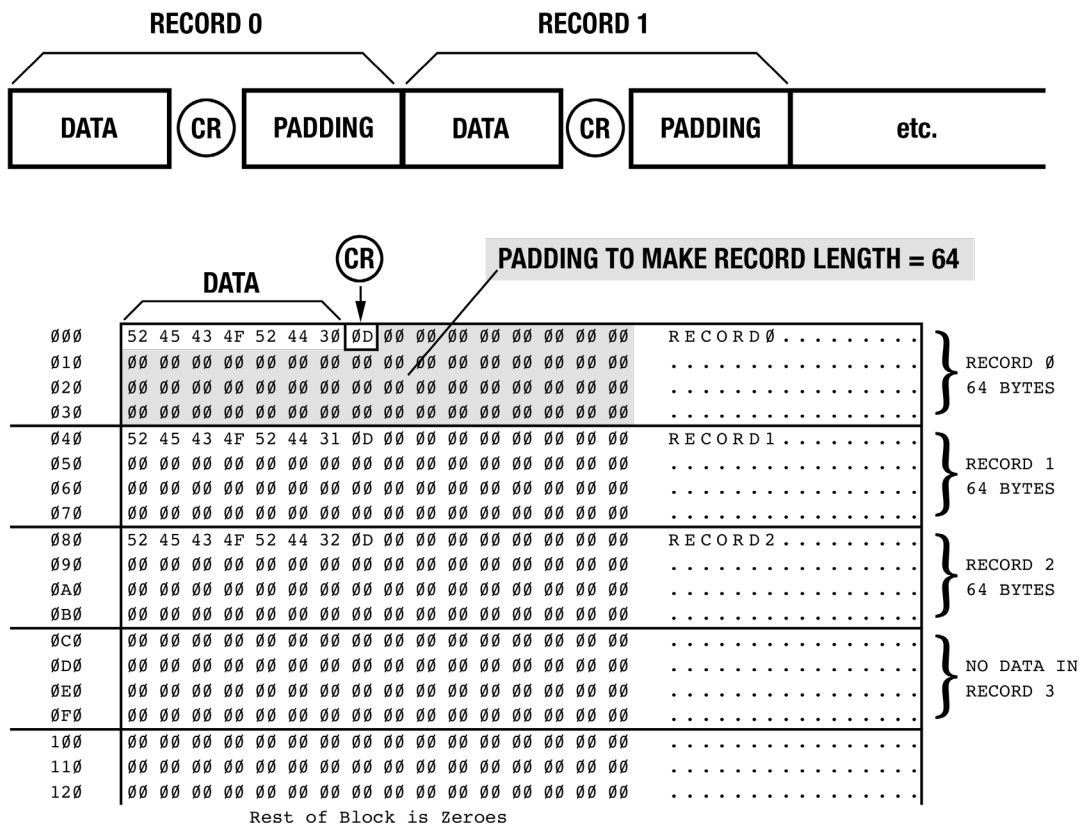
Thus, in our example, a new block is allocated to be the index block (**\$A**), another new block is allocated to be the second data block (**\$B**), both the original data block's number and the new data block's number are placed in the new index block, and, finally, the directory entry for the file is updated so that it now points to the index block instead of the seedling data block. Of course, the **STORAGE_TYPE** field in the directory entry must also be changed to indicate that this is now a **sapling** file and is no longer a seedling. Index block entries which are not associated with any data block yet (such as those beyond the end of file position) are set to zeroes. Since a block is 512 bytes long and the block numbers require a 2-byte field, this index block can store pointers to up to 256 data blocks representing up to 131072 bytes of data (128K). Obviously, most files will fall within this class of file structure. A diagram of the general form of a sapling file is given in Figure 4.7.

The index block for TXTFILE is given in Figure 4.8. Notice that the first block of the file is still block 8, the original data block of the old seedling version of TXTFILE. Notice also that in an index block, the least significant byte of the block numbers are stored in the first half of the block, and the most significant (in this example all MSB's are **\$00**) in the last half. This was done to simplify indexing into the block (the 6502 index registers can only index up to 256 bytes at a time). Thus, to find any given block, one must assemble a block number by picking the Nth byte and the N + 256th byte in the index block where N is the relative block desired.

Suppose that we now modify our program again so that 2144 records will be written. This pushes the total file size up to 137,216, more than can be described by a single index block. ProDOS must "promote" the file to the next level of the hierarchy, a **tree** file. A tree file consists of a single **master index block**, pointed to by the directory entry, which, in turn, contains the block numbers of two or more other index blocks. These lower level index blocks contain the actual data block numbers. This structure is diagrammed in Figure 4.9. Thus, since the master index block can describe 256 "subindex" blocks, and each subindex block can describe 256 data blocks, in principle this structure would support files of up to 32 megabytes! In order to limit block numbers to a 2-byte signed value of 32767, however, an arbitrary upper limit of 16 megabytes was imposed. In other words, a master index block can never be more than half full.

The entire structure for TXTFILE is depicted in Figure 4.10. Note that the original index block of the sapling file (block **\$A**) became the first subindex

A Random Text TXT Type File



AUX_TYPE in directory contains record length = 64 (\$40)

Figure 4.12 Example Random Text File Block

differs. ProDOS maintains a consistent set of file types by convention, and to a limited extent, the BASIC command interpreter enforces these conventions (e.g., “FILE TYPE MISMATCH”). You are not prevented, however, from storing an Applesoft BASIC program image in a **TEXT** file if you really work at it!

TEXT FILES

The **TEXT** or text file in its **sequential** form is the least complicated file data type (in its random form, it is, perhaps, the most complex). A sequential **TEXT** file consists of one or more records, separated from each other by carriage return characters (hex \$0Ds). This structure is shown and an example file is given in Figure 4.11. Usually, the end of a **TEXT** file is signaled by the End Of

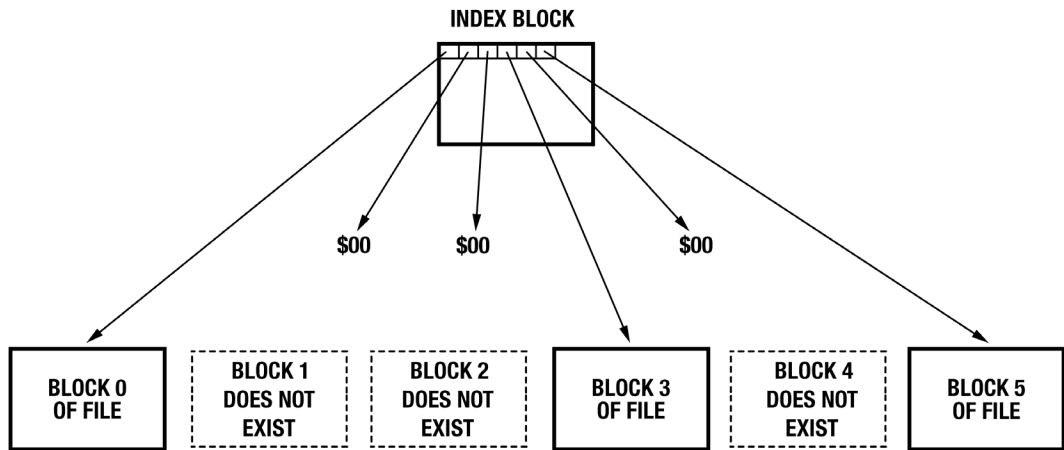


Figure 4.13 A Sparse File

File (EOF) position stored in the directory entry for the file. Since \$0D is used to delimit records, carriage returns should not appear within a record. Usually, only valid ASCII characters are allowed in a **TEXT** file to make them accessible to BASIC programs (i.e. printable text, numerics or special characters; refer to p. 8 of the *Apple II Reference Manual* or p. 16 of the *Apple II Reference Manual for IIe Only*). This restriction makes processing of a **TEXT** file slower and less efficient in the use of disk space than with a **BIN** or **VAR** type file, since each digit must occupy a full byte in the file.

When **TEXT** files are accessed **randomly**, or by record number, “holes” can appear between records. In the example given earlier and in Figure 4.12, each record is allocated 64 bytes of space in the file. By doing this, it is easy to find any record by multiplying its number by 64 and using this as a byte offset into the file. The record length is chosen as the maximum amount of space any record might occupy. Thus, records with less than 64 bytes of data, such as the ones in the example, will have wasted space at their end (filled in this case with \$00s). This wasted space is called padding. The actual data in each record is terminated with a \$0D (carriage return) just as in the sequential text file record (allowing BASIC to read it as a single **INPUT** line). In this way, data within a single record can be accessed as if it was a miniature sequential **TEXT** file. If an attempt is made to sequentially read beyond into the padding, a null string is returned.

When the randomly organized file is **OPENED**, the record length given with the “L” keyword is stored in the **AUX_TYPE** field in the directory entry for the file. Then, if later **OPENS** omit this keyword, the original value can be supplied by ProDOS.

MEMORY IMAGE...

A Binary BIN Type File

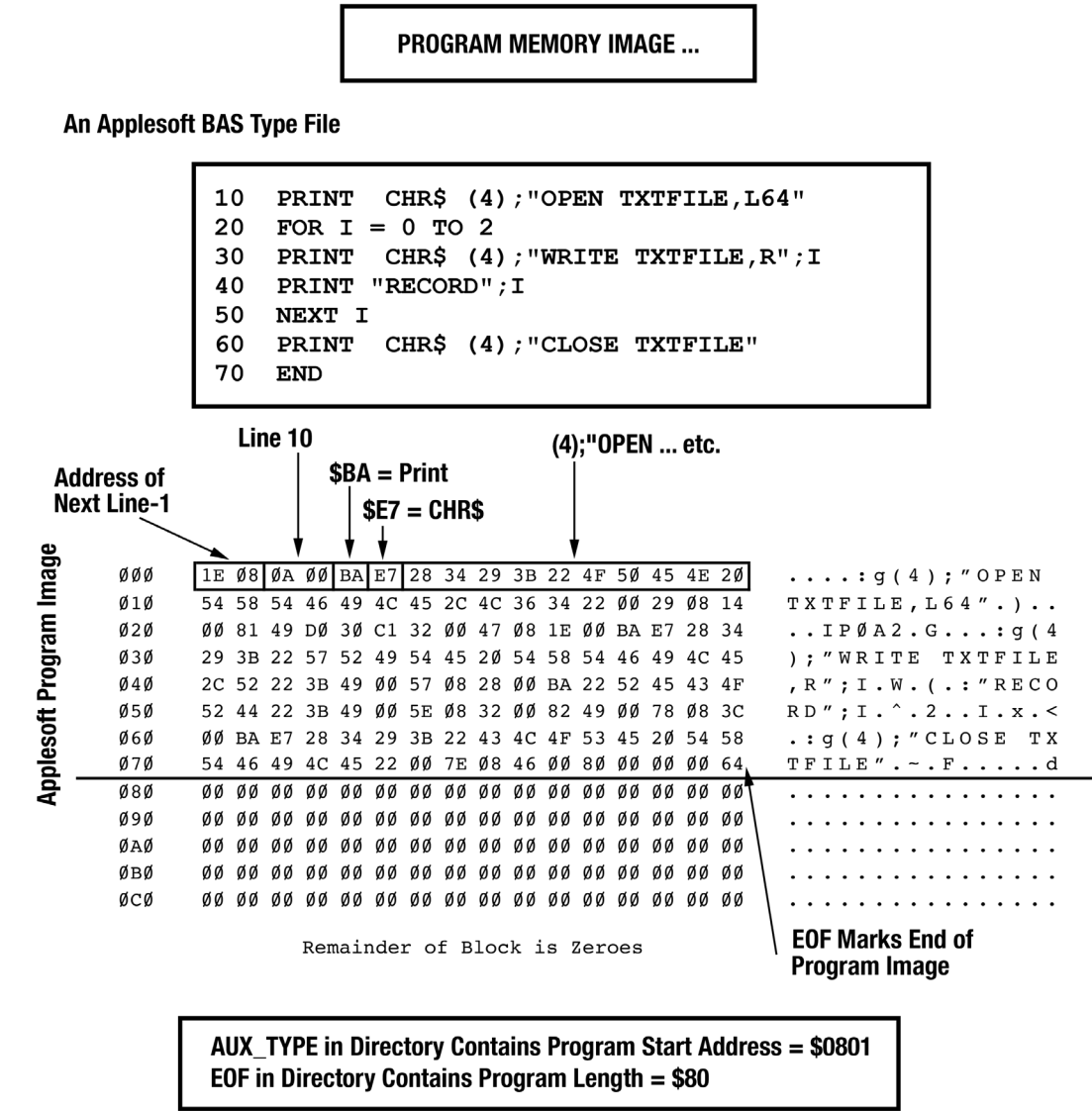
\$30 Byte Binary Memory Image																EOF Marks End of Binary Image	
00	4C	00	BE	4C	00	BE	00	00	FF	FF	00	00	FF	FF	00	00	L . > L . > . . _ _ . . _ _ . .
10	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	FF	FF	00	00	_ _ . . _ _ . . _ _ . . _ _ . .
20	59	FA	00	BE	1B	4C	03	BE	4C	00	BE	4C	59	FF	EB	BF	Y Z . > . L . > L . > L Y _ K ?
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Remainder of Block is Zeroes

AUX_TYPE in Directory Contains Address = \$03D0
 EOF in Directory Contains Length = \$30

Figure 4.14 Example BIN File Block

Notice that in the example in Figure 4.12, record 3 has not been initialized. Indeed, none of the other records following RECORD2 have anything but \$00s in them. By **WRITE**ing to specific records in a nonsequential order, it is possible to leave very large holes between records which contain data. Such files are called “sparse.” If a hole falls within a block which has other records which contain data, it is represented by binary zeroes. But if the hole covers entire blocks, ProDOS does not bother to allocate them at all. There is no point in wasting disk space on holes! Thus, if the next record containing data in our example was RECORD25, for instance, the rest of block 0 would contain zeroes (as it does now), no block would be allocated for block 1 or block 2, and block 3 would contain zeroes until the position of RECORD25 was reached. This is diagrammed in Figure 4.13. Notice that the positions of the “phantom” blocks are marked in the file’s index block with zeros. Thus, although the file covers a “data space” of six blocks, only three data blocks are actually allocated. It is possible to create a file with only two data blocks which covers the entire 16-megabyte data space. Such a file would incorporate one master index block with an entry at +0 and at +7F. All the subindex blocks in between would be “phantom,” or not allocated and marked with zero pointers. The first index block would contain a single entry at +0 for the first data block. And the last index would contain a single entry at +FF for the last data block. A 16-megabyte file using only five blocks of disk space!



AUX_TYPE in Directory Contains Program Start Address = \$0801
EOF in Directory Contains Program Length = \$80

when the command is entered, or by changing the address in the directory entry (this is sometimes necessary if the file cannot be **BSAVE**d from the location where it will run, such as from the screen buffer).

BAS FILES

A BASIC program is saved to the diskette in a way that is nearly identical to **BSAVE**. The format of a **BAS** file is given in Figure 4.15. When the **SAVE** command is typed, the ProDOS BASIC command interpreter determines the location of the BASIC program in memory and its length by examining Applesoft's zero page address. An image of the program is written to the file, and, again, the **AUX_TYPE** and **EOF** fields of the directory entry represent the address and length. Notice that the character representation of the program is somewhat garbled. This is because, in the interest of saving memory, BASIC "tokenizes" the program. Reserved BASIC words, such as **PRINT**, **IF**, **END**, or **CHR\$**, are replaced with a single hexadecimal code value (set off from other characters by its most significant bit being forced on). A complete treatment of the appearance of a BASIC program in memory is outside the scope of this manual, but a partial breakdown of the program in Figure 4.15 is given.

OTHER FILE TYPES (VAR, REL, SYS)

Several other file types have been set aside by ProDOS. Many are those found in the SOS operating system (e.g. **PCD**, **PTX**, **PDA** for Pascal, etc.). These are listed in Appendix E and will not be covered here since they are not indigenous to ProDOS. Other ProDOS file types include **BAD** and **CMD**. **BAD** files are obviously intended to mark permanent I/O errors on a disk's surface from accidental use, but there seem to be no utilities within ProDOS 1.0 which create them. The **CMD** and **PAS** file types are not currently supported by the ProDOS BASIC command interpreter, so their planned structures are anyone's guess. AppleWorks file types are designed for the AppleWorks package, and their structures are specific to that package. The formats of the **VAR**, **REL**, and **SYS** files are defined, however.

The **VAR** filetype is used to store the contents of a BASIC program's variables using the **STORE** command. The ProDOS BASIC command interpreter compresses all of the strings together with the numeric variables and saves the resulting chunk of memory as a **VAR** file. The first five bytes of the file constitute a header which defines the memory image that follows:

VAR FILE HEADER			
Byte	Offset	Length	Description
+0	(2 bytes)		Combined length of simple and array variables
+2	(2 bytes)		Length of simple variables only
+4	(1 byte)		MSB of HIMEM when these variables were STORED
+5	(n bytes)		Start of memory image...

The **AUX_TYPE** field of the directory entry for the file contains the starting address from which the compressed variables were copied. **EOF** is an indication of the end of the image. When a **RESTORE** is later issued, the memory image is reloaded, the strings are separated from the rest of the variables, and, if necessary, string pointers are adjusted based on the new **HIMEM** value.

The **REL** file type is used with a special form of binary file, containing the memory image of a machine language program which may be relocated anywhere in memory based upon additional information stored with the image

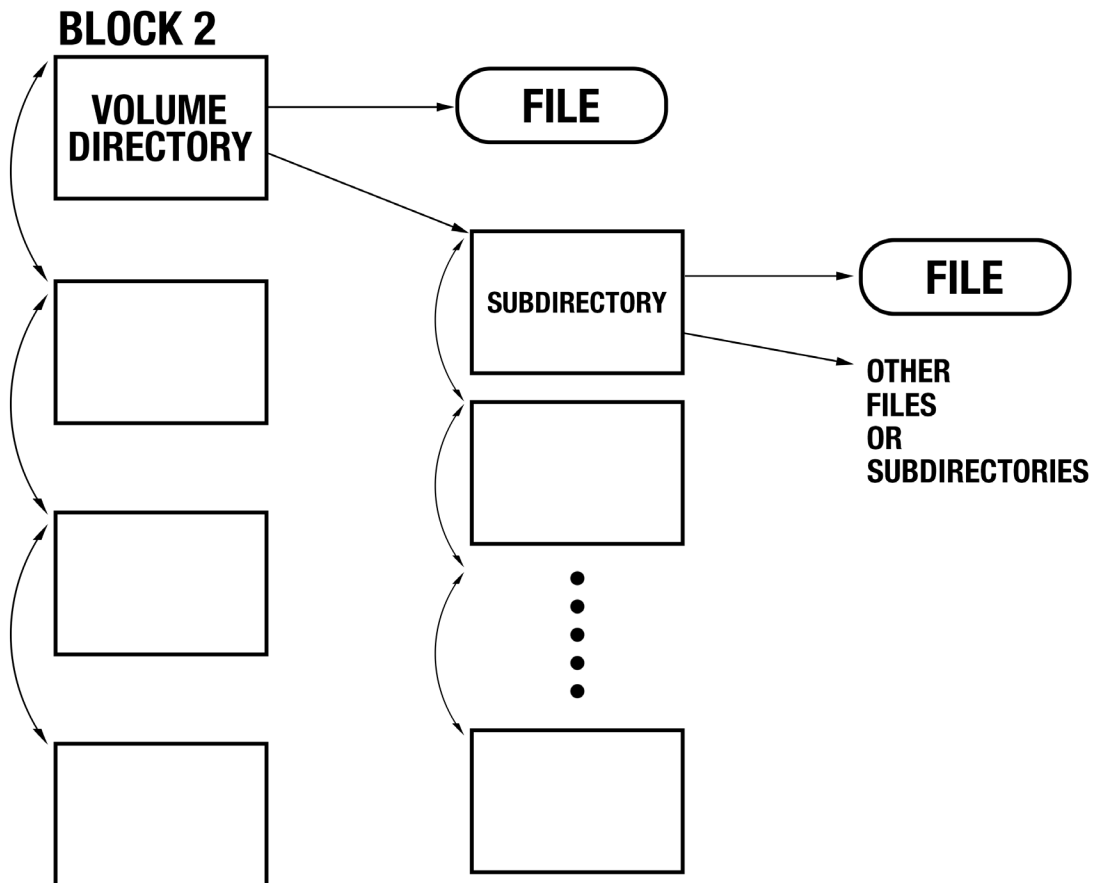


Figure 4.16 A ProDOS Subdirectory

itself. Such a file is called a Relocatable Object Module File and is produced as output from the Apple Toolkit Assembler (EDASM). The format for this type of file is given in the documentation accompanying the assembler.

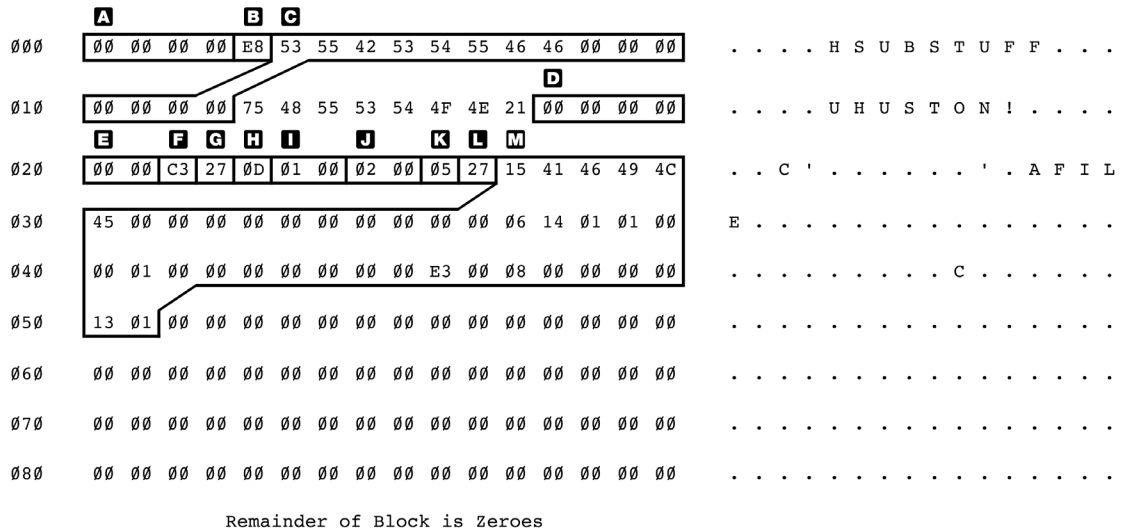
A **SYS**, or system file, is just like a **BIN** file except that it nearly always loads at \$20000 and implies a reload of the command interpreter after it exits. **SYS** files are invoked with the “-”, or smart **RUN** command, from the BASIC command interpreter. The interpreter closes all open files, frees all of the memory occupied by itself, and does a standard **BRUN** at \$20000.

DIR FILES – PRODOS SUBDIRECTORIES

Since the Volume Directory has room for just 51 entries, without subdirectories, you would be limited to 51 files per volume. This may not seem to be much of a hardship on a diskette (although it might, since DOS 3.3 allows 105), but on a hard disk with 5 million bytes or more this limit is unthinkable. In order to create a more dynamic and flexible structure, the user is permitted to create subdirectories. A subdirectory can be thought of as an extension to the Volume Directory, but there is more to it than that. In the simplest case, a subdirectory is created and an entry which describes it is placed in the Volume Directory. The subdirectory has a structure very similar to the Volume Directory: it has a header entry located at its beginning; its blocks are doubly linked by pointers in the first four bytes of each block; and it can contain file descriptive entries (including entries for “sub-subdirectories”). Unlike the Volume Directory, however, it can be of any length (it starts out with only a single block and more are added as required), its header has a slightly different format, it can be located anywhere on the diskette, and its blocks are not necessarily contiguous. A diagram of a typical subdirectory is shown in Figure 4.16. Thus, within a single subdirectory, you can create as many file entries as you have disk blocks! In practice, however, it is usually more convenient to create multiple subdirectories “dangling” from the Volume Directory, each for a specific purpose (e.g. one for word processing, one for program development, one for spreadsheets, and so on). These subdirectories might even be thought of as miniature “diskettes” within the larger volume. Although it is possible to set up very complex structures using subdirectories (multiple level tree-like networks), usually this is not very efficient or convenient and a single level (all subdirectories linked directly to the Volume Directory) works best.

One of the major concepts around which ProDOS was designed is the notion of a **path** to a file. Ordinarily, if a file is described by the Volume

SUBDIRECTORY HEADER



- A** \$0000 (previous block) \$0000 (next block) Pointer Fields (no other blocks)
- B** \$E indicates subdirectory
- C** \$8 characters in DIR name "SUBSTUFF"
- D** Directory Name padded with \$00s
- E** Creation date and time (\$0000 0000 = not set)
- F** \$00 ProDOS version that created this subdirectory
- G** \$00 Minimum version of ProDOS to access this subdirectory
- H** Access \$C3 = DESTROY / RENAME / READ / WRITE Enabled
- I** Entry Length is \$27 (39) Bytes
- J** \$0D (13) Entries per Block
- K** File Count (1 File)
- L** Parent Directory Starts in Block 2 (Volume Directory)
- M** Parent Entry Number (5th entry describes this subdirectory)
- N** Parent Entry Length is \$27 (39) Bytes
- O** File Entry for "AFILE", Seedling, Type = BIN, Data Block = \$114, EOF = \$0100, Full Access, AUX_TYPE = A\$0800, DIR Header in Block \$0113

Figure 4.17 Example Subdirectory Block

Directory, this path is very simple. ProDOS merely looks up the file in the Volume Directory and that is that. If the file is described by a subdirectory, however, ProDOS insists upon knowing how to find the subdirectory. Of course, ProDOS could systematically search all subdirectories for the file and all subdirectories of the subdirectories, and so on, but this would be very time consuming (especially if you had mistyped the file name and it didn't really exist!). Since the user usually knows which subdirectory contains the file (and, perhaps, which subdirectory describes that subdirectory, etc.) the practice is to tell ProDOS what path to follow to find a file. This is done by first specifying the volume to be searched, thereby naming the Volume Directory, followed by a list of all subdirectories which must be traversed to eventually find the file, and finally by the file name itself. For example, if in Figure 4.16 the volume name is "VOLUME" and the subdirectory is "SUB" and the file described by the subdirectory is "FILE," the path to find that file would be:

/VOLUME/SUB/FILE

If the file described by the Volume Directory in Figure 4.16 was also called "FILE" there would be no confusion at all, because its pathname would be unique:

/VOLUME/FILE

This points out an additional advantage of subdirectories. It was mentioned earlier that they were like miniature "diskettes," and, just like diskettes, there is no problem in using identical file names within different directories.

To make specifying pathnames easier, the user can specify a default prefix to ProDOS. When a file name is given (without a leading "/" in its name) it is assumed to be an incomplete pathname. To complete it, ProDOS merely attaches the prefix to the beginning. Thus, if the current prefix is:

/VOLUME/SUB/

And a reference was made to "FILE," ProDOS would create the following fully qualified pathname:

/VOLUME/SUB/FILE

Therefore, by specifying a prefix you are, in a sense, stating that you wish to work within a specific "miniature diskette," although you can still access any other file on the volume by giving its complete pathname explicitly.

An examples of a typical subdirectory block is given in Figure 4.17. The format of the Subdirectory Header is given below (remember that the first

four bytes of each subdirectory block contain the previous and next block numbers respectively):

Block Byte	Description
\$04	STORAGE_TYPE/NAME_LENGTH: The first nibble (top 4 bits) of this byte describes the type of entry. In this case, this is a Subdirectory Header so this nibble is \$E. The low 4 bits are the length of the name in the next field (the subdirectory name).
\$05-\$13	SUBDIR_NAME: A 15-byte field containing the name of this subdirectory. The actual length is defined by NAME_LENGTH above; the remainder of the field is ignored.
\$14	\$14 must contain \$75.
\$15-\$1B	Reserved for future use.
\$1C-\$1F	CREATION: The date and time of the creation of this file. This field is zero if no date was assigned. The format of the field is as follows: BYTE 0 and 1 - yyyyyyymmmdddd year/month/day BYTE 2 and 3 - 000hhhhh00mmmmmm hours/minutes where each letter above represents one binary bit. This is the standard form for all create and modify date/time stamps in directories.
\$20	VERSION: the ProDOS version number under which this subdirectory was created. This field tells later version of ProDOS not to expect to find any fields which were defined by Apple after this version of ProDOS was released. This field indicates the level of upward compatability between versions. Under ProDOS 1.0, it's value is zero.
\$21	MIN_VERSION: Minimum version of ProDOS which can access this subdirectory. A value in this field implies that significant changes were made to the field definitions since prior version of ProDOS were in use and these older versions would not be able to successfully interpret the structure of this subdirectory. This field indicates the level of downward compatability between versions. Under ProDOS 1.0, its value is zero.
\$22	ACCESS: The bits in this flag byte define how the directory may be accessed. The bit assignments are as follows: \$80 - Subdirectory may be destroyed (deleted) \$40 - Subdirectory may be renamed \$20 - Subdirectory has changed since last backup \$02 - Subdirectory may be written to \$01 - Subdirectory may be read

Block Byte	Description
	All other bits are reserved for future use.
\$23	ENTRY_LENGTH : Length of each entry in the Subdirectory in bytes (usually \$27)
\$24	ENTRIES_PER_BLOCK : Number of entries in each block of the Subdirectory (usually \$0D). Note that the Subdirectory Header is considered to be an entry.
\$25–\$26	FILE_COUNT : Number of active entries in the Subdirectory. An active entry is one which describes a file or subdirectory which has not been deleted. This count does not include the Subdirectory Header. Note that this field's name is a bit misleading since the count also includes other subdirectory entries.
\$27–\$28	PARENT_POINTER : The block number (within the volume directory or a subdirectory) which contains the file entry for this subdirectory.
\$29	PARENT_ENTRY : The number of the file entry within the block number pointed to by the PARENT_POINTER . Given that "ENTRIES_PER_BLOCK" is \$0D, then the PARENT_ENTRY number ranges from \$01 to \$0D.
\$2A	PARENT_ENTRY_LENGTH : The length of entries in the parent directory in bytes (usually \$27).

EMERGENCY REPAIRS

From time to time the information on a diskette can become damaged or lost. This can create various symptoms, ranging from mild side effects, such as the diskette not booting, to major problems, such as an input/output (I/O) error in the Volume Directory. A good understanding of the format of a diskette, as described previously, and a few program tools can allow any reasonably sharp Apple II user to patch up most errors on their diskettes.

A first question would be, "how do errors occur?" The most common cause of an error is a worn or physically damaged diskette. Usually a diskette will warn you that it is wearing out by producing "soft errors." Soft errors are I/O errors which occur randomly. You may get an I/O error message when you **CATALOG** a disk one time and have it **CATALOG** correctly if you try again. When this happens, a smart programmer immediately copies the files on the aged diskette to a brand new one and discards the old one or keeps it as a backup.

Another cause of damaged diskettes is the practice of hitting the RESET key to abort the execution of a program which is accessing the diskette. Damage will usually occur when the RESET signal comes just as data is being written onto the disk. Powering the machine off just as data is being written to the disk

is also a sure way to clobber a diskette. Of course, real hardware problems in the disk drive, cable, or controller card can cause damage as well.

If the damaged disk can be **CATALOGed**, recovery is much easier. A damaged ProDOS bootstrap loader on track 0 can usually be corrected by formatting a fresh diskette and copying the files from the old one to the new one. If only one files produces an **I/O ERROR** when it is used, it may be possible to copy most of the sectors of the file to another diskette by skipping over the bad sector with an assembler language program which calls the MLI (Machine Language Interface) in the ProDOS Kernel, or with a BASIC program (if the file is a **TXT** file). Indeed, if the problem is a bad checksum (see Chapter 3), it may be possible to read the bad sector and ignore the error and get most of the data.

An I/O error usually means that one of two conditions has occurred. Either a bad checksum was detected on the data in a sector, meaning that all bytes in the sector which follow the point of damage may be lost; or the sectoring is clobbered such that the sector no longer even exist on the diskette. If the latter is the case, the diskette (or at the very least, the track) must be reformatted, resulting in a massive loss of data. Although a program can be written to format a single track (see Appendix A), it is usually easier to copy all readable sectors from the damaged diskette to another formatted diskette and then reconstruct the lost data there.



I can't seem to find the PRODOS file around here anywhere!

Disk utilities, such as Quality Software's *Bag of Tricks*, allow the user to read and display the contents of sectors or blocks. *Bag of Tricks* will also allow you to modify the sector data and rewrite it to the same or another diskette. If you do not have *Bag of Tricks* or another commercial disk utility, you can use the **ZAP** program in Appendix A of this book. The **ZAP** program will read any block of an unprotected disk into memory, allowing the user to examine it or modify the data and then, optionally, rewrite it to a disk. Using such a program is very important when learning about diskette formats and when fixing clobbered data.

Using **ZAP**, a bad sector within a file can be localized by reading each block listed in the index blocks for that file. If the bad block is in a directory, the pointers of up to 13 files may be lost. When this occurs, a search of the diskette can be made to find "homeless" index blocks (ones which are not otherwise connected to the remaining good directory blocks in that and other directories). As these index blocks are found, new file descriptive entries can be made in the damaged sector which point to these blocks. Of course, it helps to know whether the lost files are seedlings, saplings, or trees! When the entire Volume Directory is lost, this process can take hours, even with a good understanding of the format of ProDOS volumes. Such an endeavor should only be undertaken if there is no other way to recover the data. Of course, the best policy is to create backup copies of important files periodically to simplify recovery. More information on the above procedures is given in Appendix A.

A less significant but very annoying form of diskette clobber is the loss of free blocks. It is possible, by powering off or hitting RESET at the wrong time, to leave blocks marked in use in the Volume Bit Map which were about to be marked free. These lost blocks can never be recovered by normal means, even when files are deleted, since they do not belong to anyone. The result is a **DISK FULL** message before the volume is actually full. To reclaim the lost block, it is necessary to compare every block listed in every index block or directory against the Volume Bit Map to see if there are any discrepancies. There are utility programs which will do this automatically, but the best way to solve this problem is to copy all the files on the diskette to another diskette (note that the diskette must be copied on a file by file basis, not as a volume, since a volume copy would copy an image of the diskette, bad Volume Bit Map and all).

If a file is deleted, it can usually be recovered, providing that additional block allocations have not occurred since it was deleted. If another file was created after the **DELETE** command, ProDOS probably has reused some or all of the blocks of the old file. The appropriate directory can be quickly

ZAPPED to reactivate the file (you will have to guess at the **STORAGE_TYPE** and **NAME_LENGTH** values) at **+0** in the deleted entry. The file should then be copied to another diskette and then the original deleted so that the Volume Bit Map will be correct.

FRAGMENTATION

ProDOS overhead in reading or writing blocks to a volume consists of three main parts:

1. ProDOS computational overhead time (the time to get ready to access the disk).
2. Seek time (moving the disk arm to the proper track).
3. Rotational delay (waiting for the proper sector to appear under the disk head).

In the first respect, ProDOS is an enormous improvement over Apple's earlier operating system, DOS, being up to eight times faster in its operation. This fact only increases the significance of the other two delay areas. Skewing can have an effect on rotational delay to some extent (see Chapter 3), but is much more difficult to control. Seek time, however, can vary greatly depending upon use patterns and the arrangement of files on a volume.

Imagine, for example, a volume on which a great deal of activity has occurred. Many files have been created and deleted over a period of time, leaving "holes" here and there as files are deleted, which are reallocated to existing or new files as necessary. Eventually, a map of the volume looks like a plate of spaghetti! There is nothing really wrong with this — files can be accessed normally — but if parts of an otherwise short file are spread all over the disk volume, ProDOS must spend a lot of time moving the disk read/write head from track to track to pick up all the pieces in the proper order. This costs time. A disk volume in this state of affairs is said to be badly "fragmented." Fragmentation can be even more important on a hard disk since the ratio of seek delay to rotational delay is much greater. Likewise, the best skewing setup in the world can be completely gutted by a fragmented disk, since few sequential files sectors are found together on the same track, and as the arm is moved to the new track, there is no telling how long the rotational delay will be.

When disk access time becomes a concern, it is sometimes useful to intelligently move files to specific spots on the disk. To accomplish this, the user must format a new, blank volume and copy the files from the old disk,

one by one, to the new disk in an appropriate order. Remember that ProDOS allocates blocks for files in numerically increasing order (from the outside track of the disk to the inside track). Thus, the first file you copy will be placed near the Volume Directory (a good place if you want to find that file fast). The last file you copy will go closest to the center hub of the diskette. If your program accesses two files at once, try to place them near one another on the disk. Do not separate them by many other files or you will hear the disk arm “thrashing” back and forth as it first accesses a block in file A and then must access one in file B. While you hear that noise, your program is not doing anything useful! Another thing to remember if your program opens and closes files frequently is that, when it does so, it may access several directories. It is usually a good idea in any case to keep all of your directories squashed down against the Volume Directory (i.e. **CREATE** all directories before you copy any files onto the new diskette) so that pathname searches will go faster.

Chapter 5

The Structure Of ProDOS

PRODOS MEMORY USE

ProDOS is an assembly language program which is loaded into RAM memory when the user boots the disk. Although the ProDOS machine language support routines can run by themselves in a machine smaller than 64K (or 48K plus a language card), ProDOS is primarily intended to run only on a full sized 64K or larger Apple II Plus or an Apple IIe or IIfx. In a 64K Apple II, ProDOS normally occupies the 16K of bank switched memory (or the Language Card for older Apples) and about 10.5K at the top of main memory (\$9600 through \$BFFF). The part of ProDOS which occupies the bank switched memory is called the **Kernel**. The part occupying the top of main memory is called the **BASIC Interpreter** (BI). The Kernel consists of support subroutines which may be called by any assembly language program (such as the BASIC Interpreter) to access the disk, either block by block or file by file. The BASIC Interpreter accepts ProDOS commands entered by the user or inside programs, and translates them into calls to the Kernel subroutines.* When the BASIC Interpreter is loaded, ProDOS must fool Applesoft BASIC into believing that there is actually less RAM in the machine than there is. With ProDOS loaded, Applesoft believes that there is only about 38K of RAM. ProDOS does this by adjusting **HIMEM** after it has loaded the BASIC Interpreter to prevent Applesoft from using the memory ProDOS is occupying. In order to keep track of the memory it is using, ProDOS maintains a “bit map” table which describes every page (256 bytes) in memory and marks it either free or in use. By examining this table, user written programs can avoid using previously assigned memory, even if later versions of ProDOS are loaded elsewhere.

A diagram of ProDOS's memory is given in Figure 5.1. As can be seen, there are numerous subdivisions of the two basic components mentioned above.

* It is possible, if the BASIC Interpreter's functions are not required by an application (such as a stand alone arcade-type game), to separate the Kernel from the BASIC Interpreter and not even load the BASIC Interpreter. For the purposes of this discussion, however, we will assume that ProDOS consists of both the Kernel and the BASIC Interpreter. In addition, the ProDOS Kernel may be loaded into the main part of memory if the Apple does not have a language card (48K Apple II), but the BASIC Interpreter may not be used under these circumstances because it cannot be relocated.

In addition, there are two special **global pages** containing the addresses and data pertaining to the ProDOS Kernel (SYSTEM GLOBAL PAGE at **\$BF00**) and the BASIC Interpreter (BI GLOBAL PAGE at **\$BE00**) which may be of interest to external user written programs. These global pages will be discussed in more detail later in this chapter.

As discussed earlier, ProDOS can be divided into two major components: the Kernel, containing the **Machine Language Interface** (MLI); and the BASIC Interpreter (BI). In theory, other interpreters could be written and substituted for the BI (to support Pascal or C language development, for example) but at present the only interpreter provided by Apple is the BASIC Interpreter, supporting Applesoft BASIC. There is currently no support for the older Integer BASIC language. In fact, because of the memory utilization of ProDOS, Applesoft must be resident in ROM (since the Kernel resides in the language card). Hence, ProDOS is only supported for Apple II Pluses, IIC's, and IIE's. Use of the term "BASIC Interpreter" should not be confused with the Applesoft BASIC interpreter in ROM.* Here, "interpreter" means "interpreter of disk access commands," and not "interpreter of BASIC language statements." Although the BI is closely "married" to the Applesoft interpreter in ROM, its primary responsibility is to interpret ProDOS commands which load and save files, display directories, and support file operations in BASIC programs.

The BI normally occupies memory from **\$9600** to **\$BEFF**. The first 1K (**\$9600**–**\$9A00**) is a general purpose buffer, used during Applesoft string garbage collection and for other purposes. Following this, at **\$9A00**, are the actual machine language instructions and work areas of the BI. Any data which is considered to be of interest to external programs is placed in the BI Global Page at **\$BE00**. As files are opened by BASIC programs, 1024-byte file buffers are allocated and inserted between the general purpose buffer and the BI itself. To do this, the BI must relocate the general purpose buffer and any strings which were allocated by the running BASIC program lower in memory to make room for the file buffers. **HIMEM** must be lowered accordingly. Thus, the memory available to the BASIC program fluctuates according to the number of open files.

The ProDOS Kernel occupies 12K of the 16K bank switched memory (language card). Most of the remaining 4K bank is not currently used, but is reserved by Apple for future use (the **QUIT** code occupies three pages currently). The main part of the ProDOS Kernel begins at **\$D000**, and contains the Machine Language Interface (MLI) subroutines which allow access to

* Apple's documentation also refers to the BASIC Interpreter as the "BASIC System Program."
"BASIC Interpreter" is used here because of frequent references to the "BI," an earlier designation.

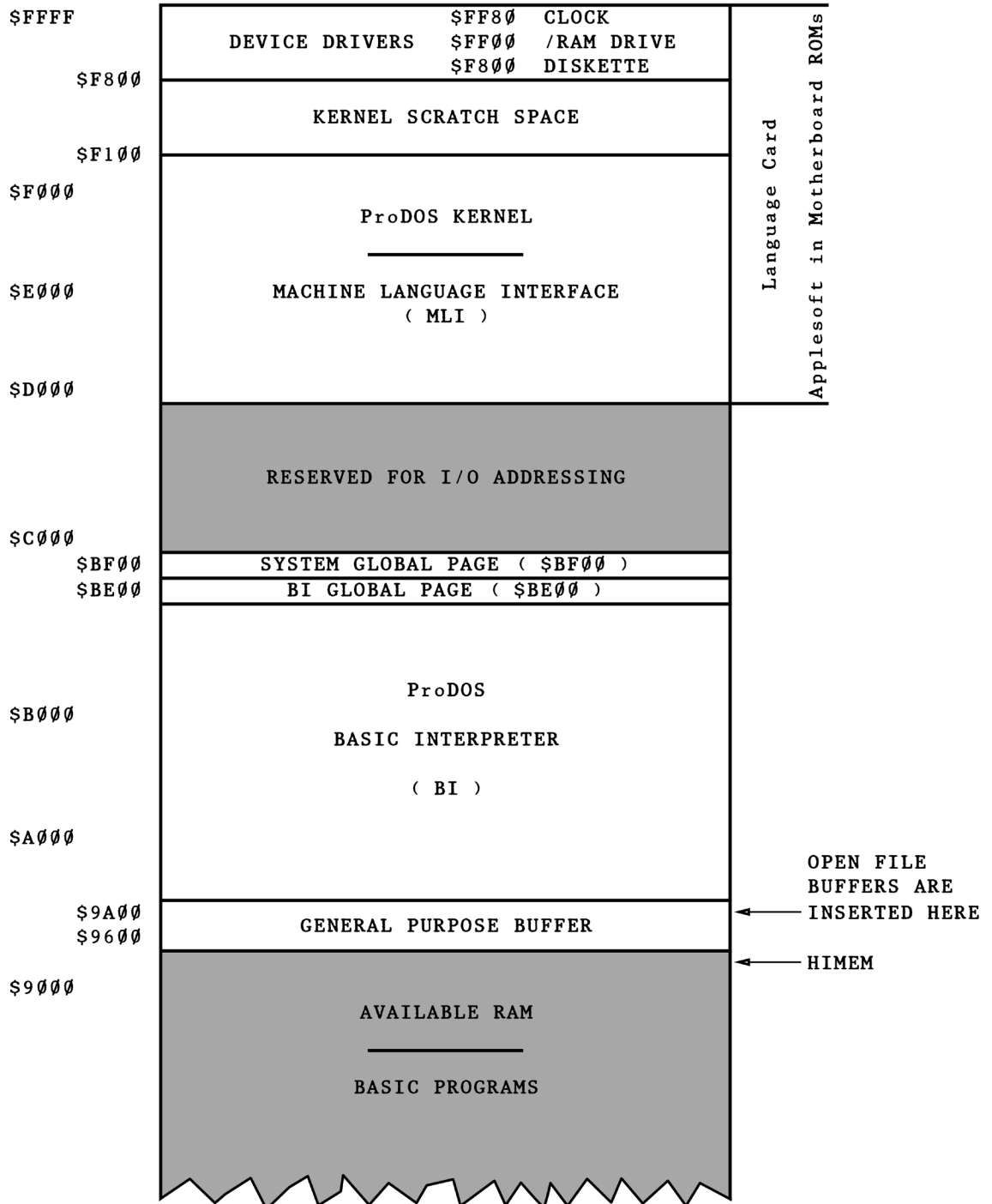


Figure 5.1 ProDOS Memory Usage (64K)

the disk by other programs (such as the BI or user written machine language programs). MLI functions provided include: open a file, create a new file, delete a file, rename a file, determine online volumes, read/write to a file, etc. The Kernel also handles interrupts for devices which can generate them. Access to these subroutines and their data is strictly controlled by the System Global Page which will be described next. Following the Kernel and its scratch space (work areas), is a 2K area devoted to device drivers. In order to provide a device independent interface to peripherals, subroutines are loaded here which can perform block oriented I/O to the Apple diskette drive, the /RAM “electronic” 64K memory diskette drive implemented in the Extended 80-Column Text card, and the Thunderclock. Additional device drivers (Hard disk, printer, etc.) must be placed in interface card ROM or in main RAM memory. The entry point addresses of each device driver in use are kept in the System Global Page.

GLOBAL PAGES

The System and BI Global Pages are maintained by ProDOS at fixed locations in main memory (**\$BF00** and **\$BE00** respectively). This practice allows important ProDOS data and subroutines to be accessed by external programs via a fixed location. Each time Apple makes a change in ProDOS and reassembles its source code, the addresses of all of the subroutines and variables may change. By putting the addresses of these routines and the variables themselves in fixed locations in memory, dependencies by a user written program on a particular version of ProDOS can be eliminated. Hopefully, all subroutines or data of general interest have been “vectored” through these global pages. If not, the programmer cannot be sure that a subroutine called directly will not move to a different memory location in a later version.

The exact format of the System Global Page is given in Chapter 8 but it contains the following information:

1. **JMP** (Goto) instructions to the main entry of the MLI, a quit vector, a clock/calendar subroutine, etc.
2. Addresses of the device drivers for each slot and drive.
3. A list of all disk drives online, and the slot and drive each occupies.
4. A “bit map” showing which pages of memory are in use and which are free.
5. Addresses of the buffers being used by MLI opened files.
6. Addresses of up to four interrupt handling routines and associated register save areas.
7. Current date, time and file level.

8. A machine ID flag byte giving the model (e.g., Apple IIe) and memory in the machine on which ProDOS is currently running.
9. Various flags indicating MLI status and whether a card occupies any slot.
10. Language card bank switching routines.
11. Interrupt entry and exit routines.
12. ProDOS version number.

The BI global page contains:

1. Addresses of routines in the BI which allow warmstart, command scanning, and error message printing.
2. I/O vectors for **PR#** and **IN#** for each slot, and the currently active input and output streams.
3. Default slot and drive.
4. BI status flags indicating whether an **EXEC** file is active, a BASIC program is running, a file is being read or written, etc.
5. Parameters that allow a user to pass an external command line to the BI.
6. A table indicating which commands allow which keyword parameters (e.g., **OPEN** does not allow the **AD** keyword but does allow the **L** keyword).
7. The current value for all keywords (**A**, **B**, **E**, **L**, **S**, **D**, etc.).
8. The address of the pathname buffers within the BI.
9. A subroutine used by the BI to access the MLI.
10. Parameter lists used by the BI to access the MLI.
11. Vectors to the BI's buffer allocate and free subroutines.
12. The current **HIMEM** MSB.

In addition to the ProDOS vectors in the global pages, the Monitor ROM and Applesoft maintain additional vectors of general interest from **\$3F0** through **\$3FF**. They are:

Location	Vector
\$3F0	LO/HI address of the routine which handles a BRK machine language instruction. Supported by the Autostart and Apple IIe and IIc ROMs. Normally contains the address of a Monitor ROM routine which prints the contents of the registers.
\$3F2	LO/HI address of routine which will handle RESET for Autostart and Apple IIe ROM. Normally the BI restart address (\$BE00) is stored here, but the user may change it you need to handle RESET 's yourself.

Location	Vector
\$3F4	Power-up byte. Contains a “funny complement” of the RESET address with an \$A5. This scheme is used to determine if the machine was just powered up or if RESET was pressed. If a power-up occurred, the Autostart ROM or Apple IIe ROM ignores the address at \$3F2 (since it has never been initialized), and attempts to boot a diskette. To prevent this from happening when you change \$3F2 to handle your own RESETs, EOR (exclusive OR) the new value at \$3F3 with an \$A5 and store the result in the power-up byte.
\$3F5	A JMP to a machine language routine which is to be called when the “&” feature is used in Applesoft. Initialized by ProDOS to point to the BI command scanner vector.
\$3F8	A JMP to a machine language routine which is to be called when a control-Y is entered from the monitor.
\$3FB	A JMP to a machine language routine which is to be called when a non-maskable interrupt (NMI) occurs.
\$3FE	LO/HI address of ProDOS’s IRQ maskable interrupt handler dispatcher. If you wish to handle an IRQ interrupt, install an interrupt handler into ProDOS—do not replace this vector.

WHAT HAPPENS DURING BOOTING

When an Apple is powered on, its memory is essentially devoid of any programs. In order to get ProDOS running, a diskette is “booted.” The term “boot” refers to the process of bootstrap loading ProDOS into RAM. Bootstrap loading involves a series of steps which load successively bigger pieces of a program until all of the program is in memory and running. In the case of ProDOS, bootstrapping occurs in two major stages, corresponding to the loading of the ProDOS Kernel and the BASIC Interpreter. Within these major stages, there are minor stages which must be performed to complete the loading process. Figures 5.2 and 5.3 diagram the processes involved in loading the Kernel and the BI respectively from the diskette. A description of this process follows.

The first boot stage is the execution of the ROM on the disk controller card. This is called the **Boot ROM**, and it exists on either the diskette controller card or a hard disk controller card at \$Cs00 (where “s” is the slot number). Thus, when the Apple is first powered on, the Monitor ROM searches the slots for a disk controller card (starting with slot 7 and moving down in slot number) and, upon finding one, it branches to \$Cs00 (usually \$C600 if the

controller is in slot 6). Control is also passed to this address should the user type **PR#6** in BASIC or **C600G** or **6 Control-P** in the monitor. The diskette controller Boot ROM is a machine language program of about 256 bytes in length. When executed, it “recalibrates” the diskette arm by pulling it back to track 0 (the “clackety-clack” noise that is heard), and then reads sector 0 from track 0 into RAM memory at location **\$800**. Once this sector has been read, the Boot ROM goes to (**JMPs**) to **\$801** which is the second stage boot, the ProDOS Loader.

The **ProDOS Loader** occupies the first block on a ProDOS diskette (physical sectors 0 and 2). Since the Boot ROM has only loaded sector 0, the

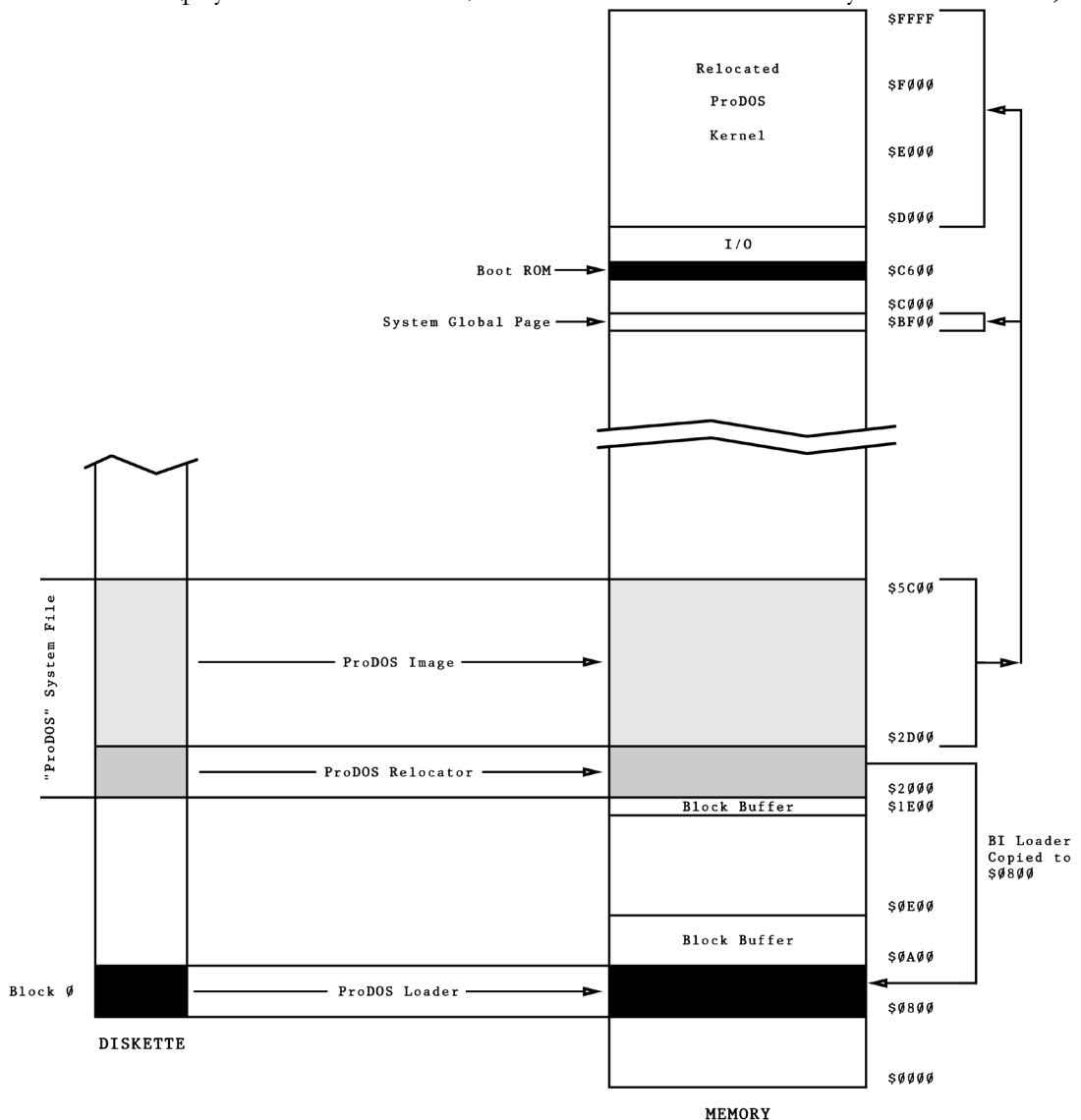


Figure 5.2 ProDOS Kernel Bootstrap Process

first task the ProDOS Loader must perform is to load the remaining sector of itself. It does this by calling the Boot ROM as a subroutine, loading it at \$9000. Having completed this, a portion of the Boot ROM is copied into a subroutine in the ProDOS Loader itself (this variable code is different for a diskette or a hard disk), and uses this to search the diskette's Volume Directory for a system file with the name "PRODOS". This file contains an image of the ProDOS Relocator, the BI Loader, and the ProDOS Kernel itself. If the file can be found, its contents are read into memory at \$20000, and the ProDOS Loader jumps to the ProDOS Relocator at \$20000.

The **ProDOS Relocator** prints a copyright and version number on the screen, and then begins to examine the machine in use to find out its model. This is done by testing the Monitor ROM for special model-dependent indicators and by checking for language card memory. The ProDOS Relocator assembles the data it has collected into a byte of flags indicating whether the machine is an Apple II, Apple II Plus, Apple IIe, Apple IIc, or an Apple III in Apple II emulation mode. It also indicates the amount of memory available. Once this has been established, the Kernel image is copied either to the bank switched memory (language card) if the machine has 64K or more, or to \$90000 for a 48K Apple. If the machine has 128K, a /RAM drive is set up in the alternate 64K memory. The peripheral card configuration is also checked, and a table of occupied slots and interface card identifications is made. The initialization of the Kernel is completed by moving an image of the System Global Page to \$BF00 and initializing it as necessary. The BI Loader image is then copied to \$8000 and control transfers there to begin booting the BI.

The **BI Loader** searches the Volume Directory for the first system file it can find whose name ends with ".SYSTEM". The file which is found will normally be BASIC.SYSTEM, but any other interpreter could be loaded in this way. If a file is found, its contents are loaded into memory at \$20000 and control passes to the BI Relocator at \$20000.

The **BI Relocator** copies the BI image to high memory (\$9A000), sets up the BI Global Page at \$BE00, and marks the pages occupied by these as "in-use" in the System Global Page's memory bitmap. The screen and keyboard vectors in zero page (CSWL/H and KSWL/H) are modified to cause immediate transfer of control to the relocater, and a jump to BASIC's coldstart entry is executed. As soon as Applesoft has completed initialization, it prints a prompt character "I ". This causes control to transfer back into the BI Relocator. CSWL/H and KSWL/H are restored to their normal settings, and initialization of the BI Global Page is completed. If a "STARTUP" file can be found in the Volume Directory, an initial command line of "-STARTUP" is dummied up

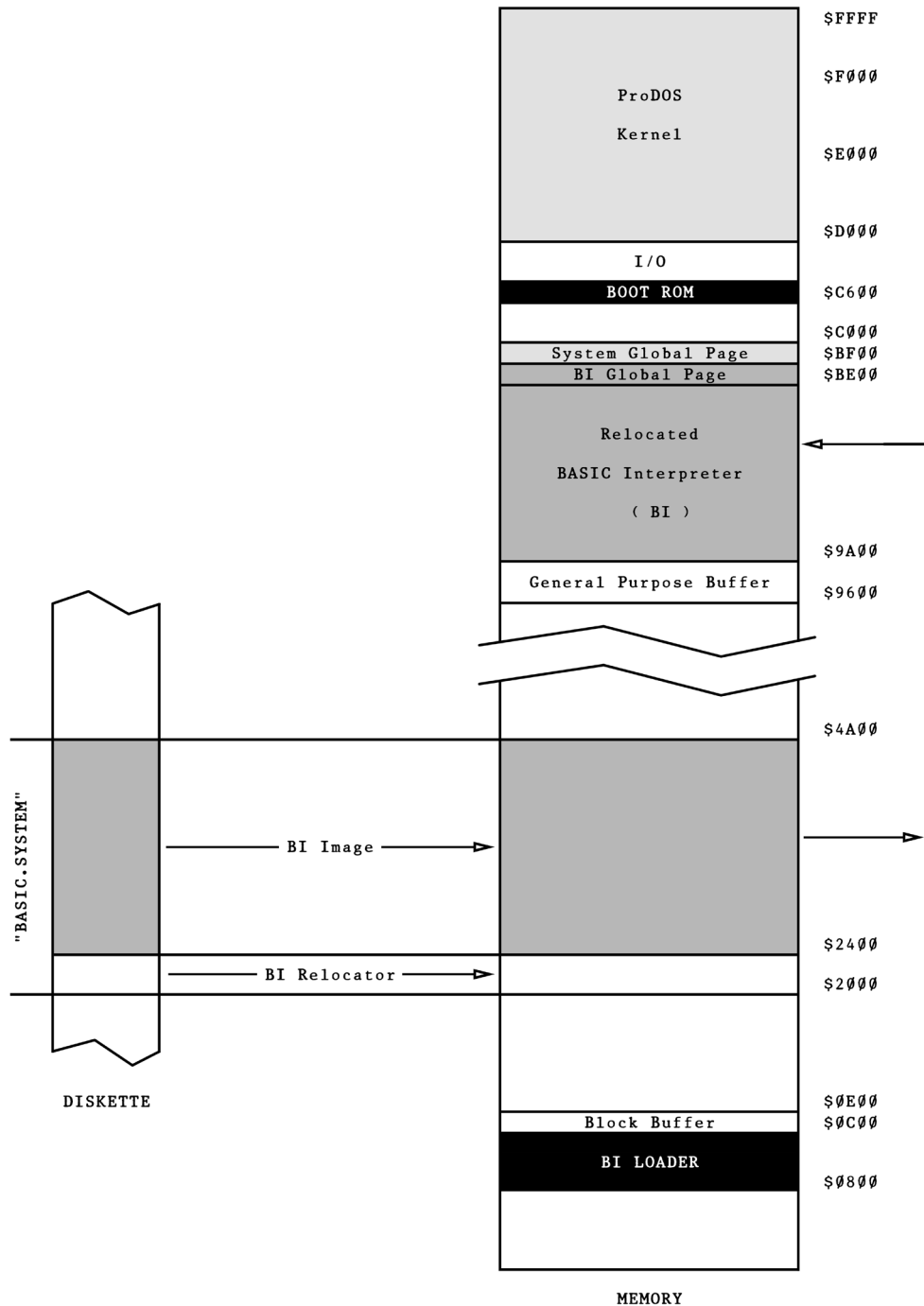


Figure 5.3 Basic Interpreter (BI) Bootstrap Process

and, after completing the vectors in page 3 (\$3F0 etc.), control transfers to the BI through its vector at \$BE00.



And then he said "HGR2:GR" and
everything went haywire

Chapter 6

Using ProDOS From Assembly Language

CAVEAT

This chapter is aimed at the advanced assembly language programmer who wishes to access the disk at any level. Access to the disk by BASIC programs is well documented in the ProDOS manual, *BASIC Programming With ProDOS*. The material presented in this chapter may be beyond the comprehension (at least for the present) of a programmer who has never used assembly language.

Access to a diskette from assembly language may be accomplished at four different levels:

Level 0	Direct access of the diskette controller
Level 1	Block access
Level 2	Machine Language Interface (MLI) access
Level 3	BI command access

At the lowest level is direct access of the diskette controller. Here, data is accessed byte by byte. This may be required to implement diskette protection schemes or to perform low level diagnostics or correction of I/O errors. The next level of access is by ProDOS blocks (two sectors per block). This is done using the appropriate ProDOS device driver; in this case, the diskette device driver. At a higher level still is the ProDOS Machine Language Interface (MLI). Here, data may be accessed on a file basis. Finally, the highest level of access is through the ProDOS BASIC Interpreter. Here, entire ProDOS command lines may be executed to produce formatted directory listings and the like. A detailed description of the programming considerations at each of these levels follows.

DIRECT USE OF THE DISKETTE DRIVE

It is often desirable or necessary to access the Apple's disk drives directly from assembly language, without the use of ProDOS. Applications which might use direct disk access range from a user written operating system to

ProDOS-independent utility programs. Direct access is accomplished using 16 addresses that provide eight on/off switches which directly control the hardware. For information on the disk hardware, please refer to Appendix D. The device address assignments are given in Table 6.1.

Switch	“OFF” Switches		“ON” Switches	
	Base Address	Function	Base Address	Function
Q0	\$C080	Phase 0 off	\$C081	Phase 0 on
Q1	\$C082	Phase 1 off	\$C083	Phase 1 on
Q2	\$C084	Phase 2 off	\$C085	Phase 2 on
Q3	\$C086	Phase 3 off	\$C087	Phase 3 on
Q4	\$C088	Drive off	\$C089	Drive on
Q5	\$C08A	Select drive 1	\$C08B	Select drive 2
Q6	\$C08C	Shift data register	\$C08D	Load data register
Q7	\$C08E	Read	\$C08F	Write

Table 6.1 ProDOS Hardware Addresses

The last two switches are difficult to explain in single phrase definitions because they interact with each other forming a 4-way switch. The four possible settings are given in Table 6.2.

Q6	Q7	Function
Off	Off	Enable read sequencing.
Off	On	Shift data register every four cycles while writing.
On	Off	Check write protect and initialize sequencer for writing.
On	On	Load data register every four cycles while writing.

Table 6.2 Four Way Q6/Q7 Switches

The addresses are slot dependent and the offsets are computed by multiplying the slot number by 16. In hexadecimal this works out nicely. Simply add the value \$s0 (where s is the slot number) to the base address. To engage disk drive number 1 in slot number 6, for example, we would add \$60 to \$C08A (device address for engaging drive 1) for a result of \$C0EA. However, since it is generally desirable to write code that is not slot dependent, one would normally use \$C08A,X (where the X-register contains the value \$s0). Table 6.3 shows the range of addresses for each slot number.

Slot Number	Address Range
0	\$C080-\$C08F
1	\$C090-\$C09F
2	\$C0A0-\$C0AF
3	\$C0B0-\$C0BF
4	\$C0C0-\$C0CF
5	\$C0D0-\$C0DF
6	\$C0E0-\$C0EF

Table 6.3 Address Ranges For Slots

In general, the above addresses need only be accessed with any valid 6502 instruction. However, in the case of reading and writing bytes (last four addresses), care must be taken to insure that the data will be in an appropriate register. All of the following would engage drive number 1. (Assume slot number 6.)

```

BIT $C0EA
LDA $C08A,X      (where X-register contains $60)
CMP $C08A,X      (where X-register contains $60)
    
```

Below are typical examples demonstrating the use of the device address assignments. For more examples, see Appendix A. All examples assume that the label SLOT is set to 16 times the desired slot number (e.g. \$60 for slot 6).

STEPPER PHASE OFF OR ON

Basically, each of the four phases (0-3) must be turned on and then off again. Done in ascending order moves the arm inward. In descending order, the arm moves outward. For optimum performance, the timing between accesses to these locations is critical, making this a nontrivial exercise. An example is provided in Appendix A demonstrating how to move the arm to a given location.

MOTOR OFF OR ON

```

LDX #SLOT          Put slot number times 16 in X-register.
LDA $C088,X        Turn motor off.
LDX #SLOT          Put slot number times 16 in X-register.
LDA $C089,X        Turn motor on (selected drive).
    
```

Note: A sufficient delay should be provided to allow the motor time to come up to speed before reading or writing to the disk. Either a specific delay or a routine that watches the data register can be used. See Appendix A for an example.

ENGAGE DRIVE 1 OR 2

LDX #SLOT	Put slot number times 16 in X-register.
LDA \$C08A,X	Engage drive 1.
LDX #SLOT	Put slot number times 16 in X-register.
LDA \$C08B,X	Engage drive 2.

READ A BYTE

	LDX #SLOT	Put slot number times 16 in X-register.
	LDA \$C08E,X	Insure read mode.
READ	LDA \$C08C,X	Put contents of data register in Accumulator.
	BPL READ	Loop until the high bit is set.

Note: \$C08E,X must be accessed to assure Read mode. The loop is necessary to assure that the accumulator will contain valid data. If the data register does not yet contain valid data, the high bit will be zero.

SENSE WRITE PROTECT

LDX #SLOT	Put slot number times 16 in X-register.
LDA \$C08D,X	
LDA \$C08E,X	Sense write protect.
BMI ERROR	If high bit set, protected.

WRITE LOAD AND WRITE A BYTE

LDX #SLOT	Put slot number times 16 in X-register.
LDA DATA	Load accumulator with byte to write.
STA \$C08D,X	Write load.
ORA \$C08C,X	Write byte.

Note: \$C08F,X must already have been accessed to insure Write mode and a 100-microsecond delay should be invoked before writing.

Due to hardware constraints, normal data bytes must be written in 32-cycle loops. The example below writes the two bytes \$D5 and \$AA to the disk. It does this by an immediate load of the accumulator, followed by a subroutine call (**WRITE9**) that writes the bytes in the accumulator. Timing is so critical that different routines may be necessary, depending on how the data is to

be accessed, and code cannot cross memory page boundaries without an adjustment.

	LDA #D5	Load byte to write. (2 cycles)
	JSR WRITE9	Go write it. (6)
	LDA #SAA	Load byte to write. (2)
	JSR WRITE9	Go write it. (6)
	...	
WRITE9	CLC	Provide different (2)
WRITE7	PHA	delays to produce (3)
	PLA	correct timing (4)
WRITE	STA \$C08D,X	Store byte in register. (5)
	ORA \$C08C,X	Write byte. (4)
	RTS	Return to caller. (6)

CALLING A STORAGE DEVICE DRIVER (BLOCK ACCESS)

ProDOS is device independent in that it requires a device driver for all storage devices. ProDOS comes with two device drivers built in. One supports the standard Apple floppy disk drive (Disk II or equivalent). The other supports a RAM drive on the Apple IIc or an Apple IIe that has 128K of memory. ProDOS can also support the ProFile hard disk which has its device driver on ROM. It seems clear that there will be many kinds of storage devices available in the future, each with its own driver.

These device drivers are used as subroutines by the MLI and provide the means of accessing the appropriate device. Four basic functions are currently defined for a device driver. They are **STATUS**, **READ**, **WRITE**, and **FORMAT**. However, not all device drivers will provide all four functions. The Disk II Device Driver, for example, does not support **FORMAT**: because of space constraints, this function is provided in the program named **FILER**.

The **READ_BLOCK** and **WRITE_BLOCK** calls in the MLI provide the only means of using a device driver from ProDOS and is the preferred method. While it is not generally recommended, any device driver can be called directly. This could prove useful in particular applications that don't require the MLI. Great care should be taken when calling the device driver directly because doing so can easily destroy data on the particular storage device.

While the parameters to call a device driver are quite straightforward, there are several potential difficulties to consider. First, RAM based device drivers normally reside in bank-switched memory, and therefore must be carefully selected and deselected. Second, a request for an unsupported device function may produce undesirable results.

There are four inputs stored in six zero page locations that must contain the appropriate information when a call is made to a device driver. The first input is the **Command Code**, which indicates which operation is requested.

As mentioned earlier, four operations are currently defined. The first of these is **STATUS**, which is used to determine if the device is ready to be accessed (either Read or Write). Although not all device drivers do so, it is suggested that the number of blocks the device supports be returned, in addition to the status. This should be done using the X (low bytes) and Y (high byte) registers. The remaining operations are quite straightforward—**READ** for reading a block, **WRITE** for writing a block, and **FORMAT** to format or initialize the media.

The second input value is the **Unit Number**, indicating in which slot and drive the desired device resides. Only two drives per slot are supported directly, but it is possible to interface a controller card that supports additional drives or volumes.

The third input is a 2-byte **Buffer Pointer** that indicates the location of a 512-byte area for data transfer. The MLI verifies that no memory conflicts exist, but most device drivers will not do so; therefore, some degree of care should be exercised in determining this input.

The fourth input is a 2-byte **Block Number** indicating which block is to be used for data transfer. The value should be in keeping with the number of blocks available on the desired device.

The four inputs necessary are listed in Table 6.4.

Although Apple has defined the manner in which device drivers are to be called, some variations will occur. Even the drivers provided by Apple vary slightly from one another. For this reason, it is advisable to make calls to any device driver with great caution. The parameter list descriptions that follow detail the four kinds of calls that are available. Not all device drivers will support all four call types and a request to an unsupported call type could prove dangerous.

Location	Description	Options
\$42	Command Code	\$00 = STATUS \$01 = READ \$02 = WRITE \$03 = FORMAT
\$43	Unit Number	DSSS0000 D=Drive number (0 = drive 1, 1 = drive 2); SSS = Slot number (0 to 7)
\$44-\$45	I/O Buffer	Can be \$0000 to \$FFFF*
\$46-\$47	Block Number	Can be \$0000 to \$FFFF
	Return Code	The processor CARRY flag is set upon return from the device driver if an error occurred. The ACCUMULATOR contains the return code. \$00 = No errors \$27 = I/O error \$28 = No device connected \$2B = Write protect error

Table 6.4 Device Driver Parameters — General Format

CALLING THE DISK II DEVICE DRIVER

Access to standard Apple floppy disk drives (Disk II or equivalent) is performed using the Disk II Device Driver provided with ProDOS. As mentioned above, the Disk II Device Driver does not support the **FORMAT** call. If such a request is made, it will be interpreted as a **WRITE** call, and serious problems may result. Formatting floppy disks is performed by a separate utility program called **FILER**.

The I/O buffer location is not checked for validity by the Disk II Device Driver. The block number must be in the range \$0-\$117 or an error type \$27 (I/O error) will result.

The Disk II Device Driver performs the same **READ** and **WRITE** functions as the RWTs routines of DOS 3.3, but these routines have been substantially modified to decrease disk access time. A comparison of RWTs to the Disk II Device Driver is contained in *Understanding the Apple IIe* by Jim Sather (1985, Quality Software).

* Unlike the MLI, the device driver will not check the location's validity.

DEVICE DRIVER PARAMETER LISTS BY COMMAND CODE

\$00 STATUS REQUEST

FUNCTION This call returns the status of a particular device and is generally used to determine if a device is present, and if so, whether it is write protected. Additionally, some drivers will return the number of blocks supported by that device.

REQUIRED INPUTS

\$42 Must be \$00.
 \$43 Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000 where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1-7).
 \$44-\$45 Unused.
 \$46-\$47 Unused but sometimes checked for validity (use \$0000).

RETURNED VALUES

Carry Flag Clear—No error occurred
 Set—Error occurred (see Accumulator for type)
 Accumulator \$00—No error occurred
 \$27—I/O error or bad block number
 \$28—No device connected to unit
 \$2B—Disk is write protected
 X-register Blocks available (low byte).
 Y-register Blocks available (high byte).

\$01 READ REQUEST

FUNCTION This call will read a 512-byte block and store it at the specified memory location. Most drivers will not check the memory location, so some care is suggested.

REQUIRED INPUTS

\$42 Must be \$01.
 \$43 Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000 where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1-7).
 \$44-\$45 Address (LO/HI) of the caller's 512-byte buffer into which the block will be read. The buffer need not be page aligned.
 \$46-\$47 Block number (LO/HI) to read. Must be valid for the device being called.

RETURNED VALUES

Carry Flag	Clear—No error occurred Set—Error occurred (see Accumulator for type)
Accumulator	\$00—No error occurred \$27—I/O error or bad block number \$28—No device connected to unit

\$02 WRITE REQUEST

FUNCTION This call will write a 512-byte block from the specified memory location. Since all write operations could potentially destroy data, care is suggested.

REQUIRED INPUTS

\$42	Must be \$02.
\$43	Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000 where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1-7).
\$44–\$45	Address (LO/HI) of the caller's 512-byte buffer which the block will be written from. The buffer need not be page aligned.
\$46–\$47	Block number (LO/HI) to read. Must be valid for the device being called.

RETURNED VALUES

Carry Flag	Clear—No error occurred Set—Error occurred (see Accumulator for type)
Accumulator	\$00—No error occurred \$27—I/O error or bad block number \$28—No device connected to unit \$2B—Disk is write protected.

\$03 FORMAT REQUEST

FUNCTION This call will format the media present in the specified device. Since all data will be destroyed, extreme care is suggested.

REQUIRED INPUTS

\$42	Must be \$03.
\$43	Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000 where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1-7).

RETURNED VALUES

Carry Flag	Clear—No error occurred
------------	-------------------------

	Set—Error occurred (see Accumulator for type)
Accumulator	\$00—No error occurred
	\$27—I/O error or bad block number
	\$28—No device connected to unit
	\$2B—Disk is write protected

CALLING THE MACHINE LANGUAGE INTERFACE

The **Machine Language Interface** (MLI) consists of a set of externally callable subroutines in the ProDOS Kernel. Over 20 different functions may be performed to access and manipulate files in a device independent manner (i.e. the programmer need not be concerned with whether the device is a diskette drive or a hard disk). To avoid duplication of code and to eliminate direct calls to unpublished entry points within ProDOS, it is recommended that all file access be performed using the standardized ProDOS Machine Language Interface.

All calls to the MLI are made through the System Global Page at **\$BF00**. The first item in this page is a **JMP** (GOTO) to the MLI. Thus, to call the MLI, code the following:

```
JSR $BF00  
DFB function_code  
DW  addr_of_parms
```

where “**function_code**” should be replaced with a 1-byte hexadecimal code representing the function you want to perform, and “**addr_of_parms**” is the 2-byte address of a parameter list you have created in your program’s memory which indicates such things as the file name being accessed, the record number to access, etc. Note that programming reentrant or “ROMable” code or routines that cannot have instructions mixed with data will be made more difficult by this convention. In these cases, it may be advisable to move the **JSR \$BF00**, the three bytes following, and a **RTS** instruction to a RAM data area and call them there.

Upon return, the processor Carry flag will be set if an error has occurred, and the return code will be placed in the A register. All other registers are saved and restored by the MLI. The valid function_codes are summarized in Table 6.5. It is interesting to note that most of the function calls are identical between ProDOS and the Apple III SOS operating system. The names used

are the standardized labels for these functions established by Apple for SOS and ProDOS.

Table 6.5 MLI Functions

Code	Name and Description
\$40	ALLOC_INTERRUPT Install interrupt handler
\$41	DEALLOC_INTERRUPT Remove interrupt handler
\$65	QUIT Exit from one Interpreter and dispatch another
\$80	READ_BLOCK Read disk block by unit number
\$81	WRITE_BLOCK Write disk block by unit number
\$82	GET_TIME Read calendar/clock peripheral card and set system date/time
\$C0	CREATE Create a new file or directory
\$C1	DESTROY Delete a file or directory
\$C2	RENAME Rename a file or directory
\$C3	SET_FILE_INFO Change a file's attributes
\$C4	GET_FILE_INFO Return a file's attributes
\$C5	ONLINE Return names of one or all online volumes
\$C6	SET_PREFIX Change default pathname prefix
\$C7	GET_PREFIX Return default pathname prefix
\$C8	OPEN Open a file
\$C9	NEWLINE

Code	Name and Description
	Set end-of-line character for line-by-line reads
\$CA	READ Read one or more bytes from an open file
\$CB	WRITE Write one or more bytes to an open file
\$CC	CLOSE Close one or more open files, flushing buffers
\$CD	FLUSH Flush all write buffers for one or more files
\$CE	SET_MARK Change File Position within an open file
\$CF	GET_MARK Return File Position within an open file
\$D0	SET_EOF Change end-of-file position of an open file
\$D1	GET_EOF Return end-of-file position of an open file
\$D2	SET_BUF Change File Buffer's address for an open file
\$D3	GET_BUF Return File Buffers address for an open file

The general form for a parameter list is as follows:

+0 PARAMETER_COUNT
+1 one or more parameters

The **PARAMETER_COUNT** is a 1-byte count of the number of parameters which follow. It is used by the MLI to validity check the parameter list to make sure that the address following the caller's **JSR** to the MLI really points to a valid parameter list.

MLI PARAMETER LISTS BY FUNCTION CODE

\$40 ALLOC_INTERRUPT INSTALL INTERRUPT HANDLER

FUNCTION

This function allows the user to install a custom interrupt handling routine into the ProDOS table. The user's handler resides in memory outside ProDOS, and only its entry point address is stored in the System Global Page table by this MLI call. Up to four such routines may be installed at any time. When a maskable interrupt (IRQ) occurs, ProDOS calls each handler in the order in which they were installed to allow the interrupt to be serviced. (See Chapter 7 for more information about writing interrupt handlers.)

PARAMETER LIST FORMAT

+0	\$02
+1	Priority
+2/+3	Address of Handler

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+2/+3	Address (LO/HI format) of user-written interrupt handling routine.

RETURNED VALUES

+1	Priority assigned to this handler by ProDOS: 1, 2, 3 or 4. This is the handler's position in the calling sequence. It is assigned the highest priority (earliest position) available.
Return Code	<p>\$00 — No errors</p> <p>\$04 — Parameter count is not \$02</p> <p>\$25 — Interrupt handler table full (4 are installed)</p> <p>\$53 — Invalid parameter in list (address is zero)</p>

\$41 DEALLOC_INTERRUPT REMOVE INTERRUPT HANDLER

FUNCTION

This function removes a previously installed interrupt handling routine's address from the ProDOS table.

PARAMETER LIST FORMAT

+0	\$01
+1	Priority

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
----	--

+1 Priority of handler to be removed (1, 2, 3, or 4) as returned by MLI call \$40 when it was installed.

RETURNED VALUES

Return Code \$00 — No errors
 \$04 — Parameter count is not \$01
 \$53 — Invalid parameter in list (PRIORITY is not 1, 2, 3, or 4)

\$65 QUIT EXIT FROM ONE INTERPRETER, DISPATCH ANOTHER

FUNCTION This function causes the MLI to move three pages of code from \$D100 in the alternate 4K of the Language card to \$1000 and branch to it. This code frees any memory allocated by the interpreter in the System Memory Bit Map in the System Global Page, and then prompts the user for the name of a new Interpreter (System Program) to be executed. It then loads the new Interpreter and executes it. For more information on this call and on writing an Interpreter, see Chapter 7.

PARAMETER LIST FORMAT

+0 \$04
 +1 Reserved
 +2/+3 Reserved
 +4 Reserved
 +5/+6 Reserved

REQUIRED INPUTS

+0 Parameter-count (4 parameters in list).
 +1 — +6 All other fields in the parameter list are reserved for future use. They must be present and they must be initialized to zeroes.

RETURNED VALUES

Return Code \$04 — Parameter count is not \$04

\$80 READ_BLOCK READ DISK BLOCK BY UNIT NUMBER

FUNCTION

This function calls the device handler for a given unit to read a 512-byte disk block. Calling this function is essentially the same as calling the device driver directly with the following additional actions: the buffer memory is validity checked for prior use; interrupts are disabled prior to the call to the driver; the unit number is validity checked and mapped into the appropriate device driver's address; the bank switched memory (language card) is enabled prior to the call and restored to its previous condition when the call completes. For these reasons, it is recommended that all block I/O be performed through the **READ_BLOCK** and **WRITE_BLOCK** MLI calls rather than calling the drivers directly. Direct calls are only recommended when the application will not be using the ProDOS Kernel and only the driver itself is available in memory.

PARAMETER LIST FORMAT

+0	\$03
+1	Unit Number
+2/+3	Address of Data Buffer
+4/+5	Block Number

REQUIRED INPUTS

+0	Parameter count (3 parameters in list).
+1	Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000, where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1 through 7).
+2/+3	Address (LO/HI) of the caller's 512-byte buffer into which the block will be read. The buffer need not be page aligned.
+4/+5	Block number (LO/HI) to read. This may range from \$0000 to \$0117 for a diskette. The validity of this number is checked by the driver itself.

RETURNED VALUES

Return Code	\$00 — No errors
	\$04 — Parameter count is not \$03
	\$27 — I/O error or bad block number
	\$28 — No device connected to unit
	\$56 — Bad buffer (already in use by ProDOS)

\$81 WRITE_BLOCK WRITE DISK BLOCK BY UNIT NUMBER

FUNCTION

This function calls the device handler for a given unit to write a 512-byte disk block. Calling this function is essentially the same as calling the device driver directly with the following additional actions: the buffer memory is validity checked for prior use; interrupts are disabled prior to the call to the driver; the unit number is validity checked and mapped into the appropriate device driver's address; the bank switched memory (language card) is enabled prior to the call and restored to its previous condition when the call completes. For these reasons, it is recommended that all block I/O be performed through the `READ_BLOCK` and `WRITE_BLOCK` MLI calls rather than calling the drivers directly. Direct calls are only recommended when the application will not be using the ProDOS Kernel and only the driver itself is available in memory.

PARAMETER LIST FORMAT

+0	\$03
+1	Unit Number
+2/+3	Address of Data Buffer
+4/+5	Block Number

REQUIRED INPUTS

+0	Parameter count (3 parameters in list).
+1	Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000, where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1 through 7).
+2/+3	Address (LO/HI) of the caller's 512-byte buffer from which the block will be written. The buffer need not be page aligned.
+4/+5	Block number (LO/HI) to write. This may range from \$00000 to \$0117 for a diskette. The validity of this number is checked by the driver itself.

RETURNED VALUES

Return Code	\$00 — No errors
	\$04 — Parameter count is not \$03
	\$27 — I/O error or bad block number
	\$28 — No device connected to unit
	\$2B — Disk is write protected
	\$56 — Bad buffer (already in use by ProDOS)

\$82 GET_TIME READ CALENDAR/CLOCK PERIPHERAL CARD

FUNCTION This function accesses any calendar/clock card which might be in the system and sets the system date and time in the System Global Page. If no calendar/clock handler has been installed (DATETIME vector in the System Global Page), the call is ignored.

PARAMETER LIST

None (parameter list address following JSR is \$0000)

REQUIRED INPUTS

None

RETURNED VALUES

\$BF90/\$BF91 System Global Page date field is filled in. Its format is (LO/HI): YYYYYYYM MMMDDDDD where YYYYYYY is the year (offset from 1900), MMMM is the month (1 through 12), and DDDDD is the day.

\$BF92/\$BF93 System Global Page time field is filled in. Its format is (LO/HI): HHHHHHHH MMMMMMMM where HHHHHHHH is the hour since midnight and MMMMMMMM is the minute (0 through 59).

Return Code \$00—No errors

\$C0 CREATE CREATE A NEW FILE OR DIRECTORY

FUNCTION This function creates a new file (either a data file or a directory file). One 512-byte block of disk space is allocated to the new file. The file may not already exist. If it is desirable to recreate an old file, issue the **DESTROY** call first. If the pathname given indicates that the file's directory entry will be in a subdirectory and there are no free directory entries there, the subdirectory will be extended by one block. The Volume Directory may not be extended. If the new file is a directory file, a directory header is created and written to the key block.

PARAMETER LIST FORMAT

+0	\$07
+1/+2	Address of Pathname
+3	Access Bits
+4	File Type
+5/+6	Auxiliary File Type
+7	Storage Type

+8/+9 Creation Date
 +A/+B Creation Time

REQUIRED INPUTS

+0 Parameter count (7 parameters in list).
 +1/+2 Address (LO/HI) of pathname buffer for file to be created. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a “/”, the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS when the file is created.

+3 Access privileges associated with this file. The access bits are: DNBXXXWR (high bit to low bit) where...

D (bit 7) if 1 allows the file to be **DESTROYED**.

N (bit 6) if 1 allows the file to be **RENAMED**.

B (bit 5) if 1 indicates file needs backing up.

X (bits 4, 3, and 2) are reserved for future use.

W (bit 1) if 1 allows the file to be written.

R (bit 0) if 1 allows the file to be read.

Full access is **\$C3**. A file is “locked” in the BASIC interpreter sense if the D, N, W and R bits are all zeroes. It is unlocked if they are all ones. The B bit is forced to one when the file is created. **WARNING:** It is possible to set the “X” reserved bits to ones with this call since no validity check is made by the MLI on **CREATE** (a check is made for **SET_FILE _INFO**, however).

+4 Type of data stored in the file. Commonly supported file types are:

\$01	BAD	File containing bad blocks.
\$04	TXT	File containing ASCII text (BASIC data file).
\$06	BIN	File containing a binary memory image or machine language program.
\$0F	DIR	File is a directory.
\$19	ADB	AppleWorks data base file
\$1A	AWP	AppleWorks word processing file
\$1B	ASF	AppleWorks spread sheet file
\$F0	CMD	ProDOS added command file.
\$F1-\$F8		User defined file types.
\$FC	BAS	File contains an Applesoft program.

\$FD	VAR	File contains Applesoft variables (STORE/RESTORE).
\$FE	REL	File contains a relocatable object module (EDASM).
\$FF	SYS	File contains a ProDOS system program.

Other less commonly used file types are defined in Appendix E. Assignment of a file type is a convention which serves to inform the program which accesses a file what data format it should expect to find there. You are not prevented from storing binary data in a **TEXT** file or ASCII text in a **BIN** file, but this runs counter to convention and is discouraged.

+5/+6 Auxiliary data pertaining to the file. Its usage is defined according to its file type above. The current uses of this field by the BI are:

TEXT contains the default record length (LO/HI).
BIN contains the address (LO/HI) at which to load the image.
BAS contains the address (LO/HI) of the BASIC program image.
VAR contains the address (LO/HI) of the BASIC variables image.
SYS contains **\$2000** (LO/HI), the load address for system files.

+7 Storage type or type of file organization. If this byte contains **\$0D**, the file is a linked subdirectory file. If it is **\$01**, it is a standard seedling file (at the time of its creation). Other values are reserved for future use. If a value of **\$00**, **\$02**, or **\$03** is given, **\$01** is assumed. All values other than **\$00-\$03** or **\$0D** will result in an error.

+8/+9 Date of creation. If this field is set to zero, the MLI uses the current system date (if any). If this field is non-zero, it is the creation date in the (LO/HI) form **YYYYYYM MMMDDDD** where **YYYYYY** is the year past 1900, **MMM** is the month (1-12) and **DDDD** is the day of the month.

+A/+B Time of creation. If this field is set to zero, the MLI uses the current system time (if any). If this field is non-zero, it is the creation time in the (LO/HI) form **HHHHHHHH MMMMMMMM** where **HHHHHHHH** is the hour past midnight and **MMMMMMMM** is the minute within the hour.

RETURNED VALUES

Return Code **\$00** — No errors
 \$04 — Parameter count is not **\$07**

- \$27—I/O error
- \$2B—Disk is write protected
- \$40—Pathname has invalid syntax
- \$44—Path to file's subdirectory is bad
- \$45—Volume directory not found
- \$47—Duplicate file name already in use
- \$48—Disk full
- \$49—Volume directory full
- \$4B—Bad storage type (use only \$0D or \$01)
- \$53—Invalid parameter or address pointer
- \$5A—Damaged disk freespace bit map

\$C1 DESTROY DELETE A FILE OR DIRECTORY

FUNCTION This function deletes a file or empty subdirectory. Open files may not be deleted. The Volume Directory may not be deleted. A subdirectory is considered “locked” if it contains any files at all, and may not be **DESTROYed** until all its files and subdirectories are **DESTROYed**.

PARAMETER LIST FORMAT

+0	\$Ø1
+1/+2	Address of Pathname

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
+1/+2	Address (LO/HI) of pathname buffer for file to be deleted. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a “/”, the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.

RETURNED VALUES

Return Code	\$00—No errors
	\$04—Parameter count is not \$01
	\$27—I/O error
	\$2B—Disk is write protected
	\$40—Pathname has invalid syntax
	\$44—Path to file's subdirectory is bad
	\$45—Volume directory not found

- \$46 — File not found in specified directory
- \$4A — Incompatible file format
- \$4B — Bad storage type
- \$4E — Access refused: **DESTROY** bit not enabled or non-empty subdirectory
- \$50 — Access refused: File is currently open
- \$5A — Damaged disk freespace bit map

\$C2 RENAME RENAME A FILE OR DIRECTORY

FUNCTION

This function renames a file or subdirectory. Only the final name in the path specification may be renamed. This function will not rename multiple directories in a pathname specification (e.g. `/project/myfile` may not be renamed to `/task/yourfile` since this involves renaming something other than the final name in the pathname). **RENAME** will not create new subdirectories or move a file's entry from one directory to another (e.g. you may not rename `/project/myfile` to `/project/another/myfile` since this involves moving the file's entry to subdirectory "another"). A volume may be renamed if no files are currently opened for it. A file or subdirectory may be renamed if it is not open, or if it is a read-only file (**WRITE** access disabled). The new file name may not be the same as another in the same directory.

PARAMETER LIST FORMAT

- | | |
|-------|-------------------------|
| +0 | \$02 |
| +1/+2 | Address of Old Pathname |
| +3/+4 | Address of New Pathname |

REQUIRED INPUTS

- | | |
|-------|--|
| +0 | Parameter count (2 parameters in list). |
| +1/+2 | Address (LO/HI) of pathname buffer for file to be renamed. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a "/", the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS. |

+3/+4 Address (LO/HI) of pathname buffer for the new name. The qualifying levels of the name, if any, should match those of the old pathname given at +1/+2. Only the final name should be different. The format of the new pathname buffer is identical to that of the old pathname buffer given above. The current default prefix, if any, will be added to a non-fully qualified pathname.

RETURNED VALUES

Return Code \$00 — No errors
 \$04 — Parameter count is not \$02
 \$27 — I/O error
 \$2B — Disk is write protected
 \$40 — Pathname has invalid syntax
 \$44 — Path to file's subdirectory is bad
 \$45 — Volume directory not found
 \$46 — File not found in specified directory
 \$47 — New name duplicates one already in directory
 \$4A — Incompatible file format
 \$4B — Bad storage type
 \$4E — Access refused: **RENAME** bit not enabled
 \$50 — Access refused: File is currently open
 \$57 — Two volumes are online with the same volume name

\$C3 SET_FILE_INFO CHANGE FILE ATTRIBUTES

FUNCTION This function changes the attributes (e.g. file type, storage type, etc.) which are stored in the directory entry which describes a file. The file may be open or closed. **SET_FILE_INFO** will not act upon a Volume Directory (an error of \$40 will result). Before issuing this function call, it is recommended that **GET_FILE_INFO** (\$C4) be used to determine the current parameter settings for the file. (Note that the parameter lists for the two calls have a compatible format.)

PARAMETER LIST FORMAT

+0 \$07
 +1/+2 Address of Pathname
 +3 Access Bits
 +4 File Type
 +5/+6 Auxiliary File Type

+7	Not Used
+8/+9	Not Used
+A/+B	Date of Last Modification
+C/+D	Time of Last Modification

REQUIRED INPUTS

+0	Parameter count (7 parameters in list).
+1/+2	Address (LO/HI) of pathname buffer for file. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a “/”, the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.
+3	New access privileges to be associated with this file. The access bits are:

DNBXXXWR

(high bit to low bit) where...

D	(bit 7) if 1 allows the file to be DESTROYED .
N	(bit 6) if 1 allows the file to be RENAMED .
B	(bit 5) if 1 indicates file needs backing up.
X	(bits 4, 3, and 2) are reserved for future use.
W	(bit 1) if 1 allows the file to be written.
R	(bit 0) if 1 allows the file to be read.

Full access is **\$C3**. A file is “locked” in the BASIC interpreter sense if the D, N, W and R bits are all zeroes. It is unlocked if they are all ones. Note that a “locked” file is not protected against **SET_FILE_INFO** (how else would one unlock it?). If an attempt is made to use the “X” reserved bits, an error will occur. They should be set to zeroes.

+4	Type of data stored in the file. Commonly supported file types are:
----	---

\$01	BAD	File containing bad blocks.
\$04	TXT	File containing ASCII text (BASIC data file).
\$06	BIN	File containing a binary memory image or machine language program.
\$0F	DIR	File is a directory.
\$19	ADB	AppleWorks data base file.
\$1A	AWP	AppleWorks word processing file.
\$1B	ASF	AppleWorks spread sheet file.
\$F0	CMD	ProDOS added command file.
\$F1–\$F8		User defined file types.

\$FC	BAS	File contains an Applesoft program.
\$FD	VAR	File contains Applesoft variables (STORE/RESTORE).
\$FE	REL	File contains a relocatable object module (EDASM).
\$FF	SYS	File contains a ProDOS system program.

Other less commonly used file types are defined in Appendix E. Assignment of a file type is a convention which serves to inform the program which accesses a file what data format it should expect to find there. You are not prevented from storing binary data in a **TEXT** file or ASCII text in a **BIN** file, but this runs counter to convention and is discouraged.

+5/+6 Auxiliary data pertaining to the file. Its usage is defined according to its file type above. The current uses of this field by the BI are:

TEXT contains the default record length (LO/HI).
BIN contains the address (LO/HI) at which to load the image.
BAS contains the address (LO/HI) of the BASIC program image.
VAR contains the address (LO/HI) of the BASIC variables image.
SYS contains \$2000 (LO/HI), the load address for system files.

+7 Ignored. May be set to zero.

+8/+9 Ignored. May be set to zero.

+A/+B Date of last modification. If this field is set to zero, the MLI uses the current system date (if any). If this field is non-zero, it is the modification date in the (LO/HI) form **YYYYYYMM MMMDDDDD** where **YYYYYY** is the year past 1900, **MMMM** is the month (1-12) and **DDDDD** is the day of the month.

+C/+D` Time of last modification. If this field is set to zero, the MLI uses the current system time (if any). If this field is non-zero, it is the modification time in the (LO/HI) form **HHHHHHHH MMMMMMMM** where **HHHHHHHH** is the hour past midnight and **MMMMMMMM** is the minute within the hour.

RETURNED VALUES

Return Code \$00 — No errors
 \$04 — Parameter count is not \$07
 \$27 — I/O error

\$2B—Disk is write protected
 \$40—Pathname has invalid syntax
 \$44—Path to file's subdirectory is bad
 \$45—Volume directory not found
 \$46—File not found in specified directory
 \$4A—Incompatible file format
 \$4B—Bad storage type
 \$4E—Access refused: Reserved access bits were used
 \$53—Parameter value out of range
 \$5A—Damaged disk freespace bit map

\$C4 GET_FILE_INFO RETURN FILE ATTRIBUTES

FUNCTION

This function reads the attributes (e.g. file type, storage type, etc.), which describe the file and are stored in the directory entry, and returns them in the parameter list provided by the caller. The file may be open or closed. If information about a Volume Directory is requested, the size of the volume in blocks and the blocks in use count are also returned.

PARAMETER LIST FORMAT

+0	\$0A
+1/+2	Address of Pathname
+3	Access Bits
+4	File Type
+5/+6	Auxiliary File Type
+7	Storage Type
+8/+9	Blocks Used
+A/+B	Date of Last Modification
+C/+D	Time of Last Modification
+E/+F	Creation Date
+10/+11	Creation Time

REQUIRED INPUTS

+0	Parameter count (\$A parameters in list).
+1/+2	Address (LO/HI) of pathname buffer for file. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a “/”, the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.

+3

New access privileges to be associated with this file. The access bits are:

DNBXXXWR

(high bit to low bit) where...

D (bit 7) if 1 allows the file to be **DESTROY**ed.

N (bit 6) if 1 allows the file to be **RENAM**ed.

B (bit 5) if 1 indicates file needs backing up.

X (bits 4, 3, and 2) are reserved for future use.

W (bit 1) if 1 allows the file to be written.

R (bit 0) if 1 allows the file to be read.

Full access is **\$C3**. A file is “locked” in the BASIC interpreter sense if the **D**, **N**, **W** and **R** bits are all zeroes. It is unlocked if they are all ones.

+4

Type of data stored in the file. Commonly supported file types are:

\$01	BAD	File containing bad blocks.
\$04	TXT	File containing ASCII text (BASIC data file).
\$06	BIN	File containing a binary memory image or machine language program.
\$0F	DIR	File is a directory.
\$19	ADB	AppleWorks data base file.
\$1A	AWP	AppleWorks word processing file.
\$1B	ASF	AppleWorks spread sheet file.
\$F0	CMD	ProDOS added command file.
\$F1–\$F8		User defined file types.
\$FC	BAS	File contains an Applesoft program.
\$FD	VAR	File contains Applesoft variables (STORE/RESTORE).
\$FE	REL	File contains a relocatable object module (EDASM).
\$FF	SYS	File contains a ProDOS system program.

Other less commonly used file types are defined in Appendix E. Assignment of a file type is a convention which serves to inform the program which accesses a file what data format it should expect to find there. You are not prevented from storing binary data in a **TXT** file or ASCII text in a **BIN** file, but this runs counter to convention and is discouraged.

+5/+6	<p>Auxiliary data pertaining to the file. Its usage is defined according to its file type above. The current uses of this field by the BI are:</p> <p>TXT contains the default record length (LO/HI).</p> <p>BIN contains the address (LO/HI) at which to load the image.</p> <p>BAS contains the address (LO/HI) of the BASIC program image.</p> <p>VAR contains the address (LO/HI) of the BASIC variables image.</p> <p>SYS contains \$20000 (LO/HI), the load address for system files.</p> <p>If the GET_FILE_INFO request is for the Volume Directory; this field contains the size of this volume in blocks</p>
+7	<p>Storage type or type of file organization. Currently supported storage types are:</p> <p>\$0D Linked directory file</p> <p>\$01 Seedling file (no index blocks)</p> <p>\$02 Sapling file (one index level)</p> <p>\$03 Tree file (two index levels)</p> <p>Other values are reserved for future use.</p>
+8/+9	<p>Number of 512-byte disk blocks in use by file including index blocks and data blocks. If the GET_FILE_INFO call is made on the volume itself (Volume Directory), this field contains the total number of disk blocks in use on the volume (including system overhead).</p>
+A/+B	<p>Date of last modification. If this field is set to zero, the MLI uses the current system date (if any). If this field is non-zero, it is the modification date in the (LO/HI) form YYYYYYMM MMMDDDDD where YYYYYY is the year past 1900, MMMM is the month (1-12) and DDDDD is the day of the month.</p>
+C/+D	<p>Time of last modification. If this field is set to zero, the MLI uses the current system time (if any). If this field is non-zero, it is the modification time in the (LO/HI) form HHHHHHHH MMMMMMMM where HHHHHHHH is the hour past midnight and MMMMMMMM is the minute within the hour.</p>
+E/+F	<p>Date of file's creation. If this field is non-zero, it is the creation date in the (LO/ HI) form YYYYYYMM MMMDDDDD where YYYYYY is the year past 1900, MMMM is the month (1-12) and DDDDD is the day of the month.</p>

+10/+11 Time of file's creation. If this field is non-zero, it is the creation time in the (LO/HI) form **HHHHHHHH MMMMMMMM** where **HHHHHHHH** is the hour past midnight and **MMMMMMMM** is the minute within the hour.

RETURNED VALUES

Return Code \$00—No errors
 \$04—Parameter count is not \$0A
 \$27—I/O error
 \$40—Pathname has invalid syntax
 \$44—Path to file's subdirectory is bad
 \$45—Volume directory not found
 \$46—File not found in specified directory
 \$4A—Incompatible file format
 \$4B—Bad storage type
 \$53—Parameter value out of range
 \$5A—Damaged disk freespace bit map

\$C5 ONLINE RETURN NAMES OF ONE OR ALL ONLINE VOLUMES

FUNCTION This function examines all mounted disk volumes and returns their names in the buffer provided by the caller. If a single volume is to be identified, the caller must provide a specific unit number (slot and drive).

PARAMETER LIST FORMAT

+0 \$02
 +1 Unit Number
 +2/+3 Address of Data Buffer

REQUIRED INPUTS

+0 Parameter count (2 parameters in list)
 +1 Unit number of specific device to be examined. If all online volumes are to be identified, set this field to zero. The bit assignment for a specific unit number is DSSS0000, where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1 through 7)
 +2/+3 Address (LO/ HI) of a buffer to contain the volume names returned by ProDOS. If a specific unit is to be examined a 16-byte buffer must be provided. If the call is non-specific (UNIT = 0), then the buffer must be 256 bytes to allow for up to 16 online volumes.

RETURNED VALUES

Buffer

If the return code in the accumulator is zero, the caller's buffer will contain zero or more volume name entries of format described below. The volume names will be given in the order in which ProDOS searches for a volume, i.e., the boot volume first, followed by slot numbers lower than the boot slot, wrapping around to higher slots last.

Online Volume Entry

byte 0	DSSSLLLL: where D is the drive number (0=drive 1, 1=drive 2), SSS is the slot number (1 through 7), and LLLL is the length of the name which follows. If LLLL is zero, an error occurred in examining this volume. The return code is in the first byte of the name field. If byte 0 is zero, then there are no more volume entries in the buffer.
bytes 1-15	Volume name or 1-byte error code. No slash precedes the name.

Return Code

\$00 — No errors

\$04 — Parameter count is not **\$02**

\$55 — Volume Control Block full (too many open files)

\$56 — Bad buffer address (check system memory bit map)

The following error codes may appear for a specific unit in byte 1 of a buffer entry. If so, the return code above will be **\$00**.

\$27 — I/O error on this unit

\$28 — Device not connected (e.g. no drive 2)

\$2E — Diskette switched while file was open

\$45 — Volume directory not found

\$52 — Not a ProDOS disk volume

\$57 — Duplicate volume — Byte 3 of buffer entry contains the unit number of the duplicate

\$C6 SET_PREFIX CHANGE DEFAULT PATHNAME PREFIX

FUNCTION

This function changes the default prefix which is attached to any pathnames passed to the MLI which are not fully qualified (do not start with a slash “/”). The MLI follows the prefix given, locating each directory at each level of the prefix to make sure that they exist on a mounted volume.

PARAMETER LIST FORMAT

+0	\$Ø1
+1/+2	Address of Pathname Buffer

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
+1/+2	Address (LO/HI) of pathname buffer for the new prefix. The pathname buffer consists of one byte of length followed by 1 to 63 characters of name. If the first character is a “/”, the name is considered to be fully qualified. If not, the old default prefix is added to the new one to form a completely qualified default prefix (for a total length of no more than 64 characters). The last name in the prefix must be that of a directory file. The prefix may be eliminated by specifying a null (0-length) prefix. An ending slash is assumed if it is omitted.

RETURNED VALUES

Return Code	
\$00	—No errors
\$04	—Parameter count is not \$01
\$40	—Pathname has invalid syntax or prefix too long
\$44	—Path to final subdirectory is bad
\$45	—Volume directory not found
\$46	—Final subdirectory file not found
\$4A	—Incompatible file format
\$4B	—Bad storage type
\$5A	—Damaged disk freespace bit map

\$C7 GET_PREFIX RETURN DEFAULT PATHNAME PREFIX

FUNCTION This function returns the default prefix, if any, to the caller's buffer.

PARAMETER LIST FORMAT

+0	\$01
+1/+2	Address of Pathname

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
+1/+2	Address (LO/HI) of pathname buffer into which the MLI will copy the default prefix. The buffer must be at least 64 bytes long.

RETURNED VALUES

Buffer	The buffer will contain the current MLI default prefix. The prefix consists of one byte of length followed by up to 63 characters of prefix. If the length is zero, the prefix is null. Otherwise, the prefix starts and ends with a slash.
Return Code	<p>\$00 — No errors</p> <p>\$04 — Parameter count is not \$01</p> <p>\$56 — Bad buffer address (check system memory bit map)</p>

\$C8 OPEN OPEN A FILE

FUNCTION This function locates a file on a volume and sets up internal control blocks (a File Control Block—FCB, and a Volume Control Block—VCB) to allow the user to read or write it. A reference number (from 1 to 8) is assigned by the MLI to the open file for future identification. (The reference number uniquely identifies the FCB which is being used with the file.) The current position for reading or writing is set to zero (start of the file). At most, eight files may be open at one time. More than one **OPEN** may be issued to the same file if the file's access is **WRITE** disabled (read-only file).

Once a file is opened, it should always be closed (using the MLI **CLOSE** call). This is to permit the MLI to release the reference number for use by other **OPENs**. In addition, the MLI keeps a count of the number of files which are open on a volume. If the diskette is switched while files are open, error return codes are produced.

A directory file may also be opened (for **READS** only). When accessing a directory, do not make assumptions about the length of an entry or the number of entries per block--use the fields in the directory header which are provided for this purpose. This will help to insure that your program will work for future releases of ProDOS. A directory file may be read only, not written.

PARAMETER LIST FORMAT

+0	\$03
+1/+2	Address of Pathname
+3/+4	Address of File Buffer
+5	Reference Number

REQUIRED INPUTS

+0	Parameter count (3 parameters in list).
+1/+2	Address (LO/HI) of pathname buffer for file. The pathname buffer consists of one byte of length followed by 1 to 63 characters of name. If the first character is a “/”, the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.
+3/+4	Address (LO/HI) of a 1024-byte file buffer, provided by the caller in his memory, to be used by the MLI while the file is open. The buffer must begin on an even page boundary (LO portion of address must be zero). The MLI uses the buffer to hold the current data block and the current index block respectively. Its contents need not be initialized by the caller. It should not be tampered with by the caller while the file remains open.
\$BF94	The LEVEL byte in the System Global Page may be set to indicate the level of this OPEN . If a subsequent CLOSE is issued with a REF_NUM of zero, then all files of a given level or higher will be closed. This feature is handy in that it allows group CLOSEs on user-defined classes of files. Normally, LEVEL is set to zero.

RETURNED VALUES

+5	A reference number assigned to this open file by the MLI (from \$01 to \$08). The caller should make a note of this number and use it in all future references to this open file. A reference number is used to identify open files instead of the pathname since it is possible to maintain multiple “opens” on the same read-only file.
Return Code	\$00 — No errors \$04 — Parameter count is not \$03

- \$27 — I/O error
- \$40 — Pathname has invalid syntax
- \$42 — Eight files are already open
- \$44 — Path to file's subdirectory is bad
- \$45 — Volume directory not found
- \$46 — File not found in specified directory
- \$4B — Bad storage type
- \$50 — File already open (**WRITE** enabled)
- \$53 — Parameter value out of range (**REF_NUM**)
- \$56 — Bad buffer address (check system memory bit map)
- \$5A — Damaged disk freespace bit map

\$C9 NEWLINE SET END OF LINE CHARACTER

FUNCTION

A file may be read as either a continuous stream of bytes or as a collection of lines, terminated by “newline” characters (such as a **RETURN** character). When a file is first opened, the former assumption is made. To enable the line by line mode, the **NEWLINE** function may be invoked, specifying the end of line character to be used. All future **READ** operations on the specified open file will be terminated either when a newline character is detected, or when the read length is exhausted (or at end of file).

PARAMETER LIST FORMAT

- | | |
|----|-------------------|
| +0 | \$03 |
| +1 | Reference Number |
| +2 | AND Mask |
| +3 | Newline Character |

REQUIRED INPUTS

- | | |
|----|--|
| +0 | Parameter count (3 parameters in list). |
| +1 | Reference number for an open file as returned by OPEN . |

- +2 **AND** mask. The value given here is logically **AND**ed with the contents of each byte read before a comparison is made with the **NEWLINE** character given in +3. If the **AND** mask is zero, then the line by line mode is disabled and the continuous byte stream mode is enabled. If a mask of **\$FF** is given, the **NEWLINE** character must exactly match what is read. Other values for the **AND** mask allow “don’t care” bits. For example, **\$7F** allows the MSB to be either on or off without affecting the comparison (e.g., **\$0D** or **\$8D** will both be treated as newline if **\$0D** is the **NEWLINE** character and the **AND** mask is **\$7F**).
- +3 The actual value of the **NEWLINE** character. Normally, when line by line mode is used, this should be set to **\$0D**. Note that if the **AND** mask is **\$00**, this character is ignored (even if it is also **\$00**; if **\$00** is to be the newline character, set the **AND** mask to **\$FF**).

RETURNED VALUES

- Return Code **\$00** — No errors
 \$04 — Parameter count is not **\$03**
 \$43 — Invalid reference number

\$CA READ READ ONE OR MORE BYTES FROM AN OPEN FILE

FUNCTION

This function reads a number of bytes, starting at the current file position in an open file. The number of bytes read depends upon the length requested by the caller, whether or not a newline character has been set (see the **\$C9 NEWLINE** function call), and whether the end of file is reached during the read. The current file position is updated to point to the byte following the last byte read.

In general, read operations will be much more efficient if the amount of data transferred exceeds a block (512 bytes). Special “direct read” code exists within the MLI to prevent “double buffering” and allow direct reads to the caller’s buffer without going through the I/O buffer attached to the file. This fast access is only used when whole blocks may be read at a time. Use of the **NEWLINE** feature automatically disables “direct reads.” (NOTE: It is this “direct read” feature which makes ProDOS I/O faster than Apple DOS.)

Note that, once a file is opened, no check is made by the MLI that the user has not switched diskette volumes in the drive. If this occurs, it is possible to read random portions of the new diskette volume! If the programmer is issuing a **READ** after a period of disk inactivity, it is recommended that a **ONLINE** call (\$C5) be issued to make sure that the same diskette is still in the drive.

PARAMETER LIST FORMAT

+0	\$04
+1	Reference Number
+2/+3	Address of Data Buffer
+4/+5	Requested Length
+6/+7	Actual Length

REQUIRED INPUTS

+0	Parameter count (4 parameters in list).
+1	Reference number for an open file as returned by OPEN .
+2/+3	Address (LO/HI) of a sufficiently large buffer provided by the caller into which the data will be read. This buffer should not be confused with the “file buffer” passed to OPEN which is separate, and should not be used by the caller’s program.
+4/+5	Maximum number (LO/HI) of bytes of data to read. This is usually the size of the data buffer. If lines are being read, make sure this value is as large as the longest line, including the end of line character itself.

RETURNED VALUES

+6/+7	Actual number (LO/HI) of bytes placed in the caller’s data buffer by the MLI. This value will differ from the requested length in +4/+5 if a newline character was found, if the end of the file was reached, or if an error occurred during the read operation. If a newline character terminated the read, this length will include the newline character itself. If the read began at the end of file position, this field is set to zero, and the end of file return code (\$4C) is placed in the A register.
Return Code	<p>\$00 — No errors</p> <p>\$04 — Parameter count is not \$04</p> <p>\$27 — I/O error</p> <p>\$43 — Invalid reference number</p> <p>\$4C — At end of file, nothing was read</p> <p>\$4E — Access refused: Read bit not enabled</p> <p>\$56 — Bad buffer address (check System memory bit map)</p> <p>\$5A — Damaged disk freespace bit map</p>

\$CB WRITE

WRITE ONE OR MORE BYTES TO AN OPEN FILE

FUNCTION

This function writes a number of bytes to disk, starting at the current file position in an open file. You may not write to a directory. The current file position is updated to point to the byte following the last byte written. The end of file mark is moved if necessary, and new data and/or index blocks are allocated to the file as necessary. In the interest of efficiency, the data may or may not be written to disk at this time. As much as one block's worth (512 bytes) may remain in the file buffer to be written later when the block is filled, the file is closed or flushed, or when the file position is changed. For this reason, it is important to close all files before powering off the machine.

Note that, once a file is opened, no check is made by the MLI that the user has not switched diskette volumes in the drive. If this occurs, it is possible to Write on random portions of the new diskette volume! If the programmer is issuing a **WRITE** after a period of disk inactivity, it is recommended that a **RETURN ONLINE** volumes call (\$C5) be issued to make sure that the same diskette is still in the drive.

Note that there is no “direct write” feature similar to the “direct read” feature described under the **READ** MLI call.

PARAMETER LIST FORMAT

+0	\$Ø4
+1	Reference Number
+2/+3	Address of Data Buffer
+4/+5	Requested Length
+6/+7	Actual Length

REQUIRED INPUTS

+0	Parameter count (4 parameters in list).
+1	Reference number for an open file as returned by OPEN .
+2/+3	Address (LO/HI) of the data to be written to disk. This buffer should not be confused with the “file buffer” passed to OPEN which is separate, and should not be used by the caller's program.
+4/+5	Number (LO/HI) of bytes of data to write from the data buffer.

RETURNED VALUES

+6/+7	Actual number of bytes written. Unless an error occurs during the operation, this field should match the requested length in +4/+5.
Return Code	<p>\$00 — No errors</p> <p>\$04 — Parameter count is not \$04</p> <p>\$27 — I/O error</p> <p>\$2B — Disk is write protected</p> <p>\$43 — Invalid reference number</p> <p>\$48 — Disk full</p> <p>\$4E — Access refused: WRITE bit not enabled</p> <p>\$56 — Bad buffer address (check System memory bit map)</p> <p>\$5A — Damaged disk freespace bit map</p>

\$CC CLOSE CLOSE OPEN FILE(S), FLUSHING BUFFERS

FUNCTION

For a specific open file, this function flushes any data which has not yet actually gone to disk from the file buffer, releases the file buffer to the caller for reuse, sets the **BACKUP** bit in the **ACCESS** flags for the file, updates the directory entry for the file with block count, etc., and frees the reference number and File Control Block (FCB) for use with a later **OPEN**. Each **OPEN** must have a corresponding **CLOSE**. If a non-specific call is made (**REFNUM** = 0), all open files at the current **LEVEL** (\$BF94) or higher are closed.

PARAMETER LIST FORMAT

+0	\$01
+1	Reference Number

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
+1	Reference number for an open file as returned by OPEN or \$00 if all files at the current level or higher are to be closed. If a multiple file request is made and an error occurs on one file, this does not prevent the MLI from attempting to complete the close operation for any other files. If multiple errors occur, only the last error return code is passed back to the caller.

\$BF94	Current file LEVEL in the System Global Page. If set to \$00 before this call, all open files are closed.
--------	--

RETURNED VALUES

Return Code	\$00—No errors
	\$04—Parameter count is not \$01
	\$27—I/O error
	\$2B—Disk is write protected
	\$43—Invalid reference number
	\$5A—Damaged disk freespace bit map

\$CD FLUSH FLUSH ALL WRITE BUFFERS FOR FILES

FUNCTION

For a specific open file, this function flushes any data which has not yet actually gone to disk from the file buffer, updates the directory entry for the file, and sets the **BACKUP** bit in the **ACCESS** flags for the file (if data was written). If no write operations have occurred, then the **FLUSH** call is ignored. If a non-specific call is made (**REFNUM** = 0), all open files at the current **LEVEL** (\$BF94) or higher are flushed. The flush call is useful when it is desirable to force write data out to disk before a long period of inactivity in case of power loss or other disasters.

PARAMETER LIST FORMAT

+0	\$01
+1	Reference Number

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
+1	Reference number for an open file as returned by OPEN , or \$00 if all files at the current level or higher are to be flushed. If a multiple file request is made and an error occurs on one file, this does not prevent the MLI from attempting to complete the flush operation for any other files. If multiple errors occur, only the last error return code is passed back to the caller.
\$BF94	Current file LEVEL in the System Global Page. If set to \$00 before this call, all open files are flushed.

RETURNED VALUES

Return Code	\$00—No errors
	\$04—Parameter count is not \$01
	\$27—I/O error
	\$2B—Disk is write protected

\$43 — Invalid reference number

\$5A — Damaged disk freespace bit map

\$CE SET_MARK CHANGE FILE POSITION WITHIN AN OPEN FILE

FUNCTION

When a file is first opened, the MLI establishes a “file position” at which reading or writing will occur at the beginning of the file (zero). As data is read or written, the file position is moved to allow sequential access to the file. This file position describes the relative byte offset to the next byte in the file to be accessed. If random access to a file is desired, the caller may use this function to change the position to another location in the file before issuing a **READ** or **WRITE** call. If the file position is moved to an area of the file where no data exists (i.e. an area which has never been written), new data and/or index blocks will be allocated when the next **WRITE** call is made. This function may be used in conjunction with the **GET_EOF** call (\$D1) to append data to the end of a file.

PARAMETER LIST FORMAT

+0	\$02
+1	Reference Number
+2/+3/+4	New File Position

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1	Reference number for an open file as returned by OPEN .
+2/+3/+4	The new file position to be set. This is a 3-byte number (least significant byte first, most significant byte last) representing the byte offset into the file. The position of the first byte in a file is zero. The position may not exceed the current end of file position.

RETURNED VALUES

Return Code	\$00 — No errors
	\$04 — Parameter count is not \$02
	\$43 — Invalid reference number
	\$4D — File position beyond end of file
	\$5A — Damaged disk freespace bitmap

\$CF GET_MARK RETURN FILE POSITION WITHIN AN OPEN FILE

FUNCTION When a file is first opened, the MLI establishes a “file position” at which reading or writing will occur at the beginning of the file (zero). As data is read or written, the file position is moved to allow sequential access to the file. This file position describes the relative byte offset to the next byte in the file to be accessed. This function will return the current value of the file position.

PARAMETER LIST FORMAT

+0	\$02
+1	Reference Number
+2/+3/+4	File Position

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1	Reference number for an open file as returned by OPEN .

RETURNED VALUES

+2/+3/+4	The current file position value. This is a 3-byte number (least significant byte first, most significant byte last) representing the byte offset into the file of the next byte to be read or written. The position of the first byte in a file is zero.
----------	--

Return Code	\$00 — No errors
	\$04 — Parameter count is not \$02
	\$43 — Invalid reference number

\$D0 SET_EOF CHANGE END OF FILE POSITION OF AN OPEN FILE

FUNCTION

This function changes the end of file mark (or file size). It is not normally necessary to change the end of file mark since the **WRITE** function will automatically extend the **EOF** mark as new data is written to the end of the file. This function is useful, however, to truncate a file or to allow random positioning within a very large sparse file. If the new end of file position passed by the caller is less than the old one, the file is truncated and excess data and index blocks are freed for reuse by the system. If it exceeds or equals the old value, no new blocks will be allocated until they are needed in a **WRITE** operation. If the new end of file would leave the current file position outside the limits of the file, it is forced back to the new end of file position. The **EOF** mark of a directory file may not be changed with **SET_EOF**. Note that the file size does not necessarily represent the amount of disk space the file requires, since the file may be sparse (see Chapter 4).

PARAMETER LIST FORMAT

+0	\$02
+1	Reference Number
+2/+3/+4	New EOF Position

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1	Reference number for an open file as returned by OPEN .
+2/+3/+4	The new end of file position. This is a 3-byte number (least significant byte first, most significant byte last) representing the byte offset into the file of the last byte plus one. The position of the first byte in the file is zero (the EOF of an empty file).

RETURNED VALUES

Return Code	\$00 — No errors
	\$04 — Parameter count is not \$02
	\$27 — I/O error
	\$43 — Invalid reference number
	\$4D — Position is too large for volume
	\$4E — Access refused: WRITE bit not enabled
	\$5A — Damaged disk freespace bit map

\$D1 GET_EOF RETURN END OF FILE POSITION OF AN OPEN FILE

FUNCTION This function returns the value of the end of file mark for an open file. `GET_EOF` may be used to determine the size of a sequential file or to find the end of a file so that data may be appended to it. `GET_EOF` for a directory file will return the number of blocks used multiplied by 512 bytes. Note that the file size does not necessarily represent the amount of disk space the file requires, since the file may be sparse (see Chapter 4).

PARAMETER LIST FORMAT

+0	\$02
+1	Reference Number
+2/+3/+4	EOF Position

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1	Reference number for an open file as returned by <code>OPEN</code> .

RETURNED VALUES

+2/+3/+4	The current end of file position. This is a 3-byte number (least significant byte first, most significant byte last) representing the byte offset into the file of the last byte plus one. The position of the first byte in the file is zero (the <code>EOF</code> of an empty file).
----------	--

Return Code	\$00 — No errors
	\$04 — Parameter count is not \$02
	\$43 — Invalid reference number

\$D2 SET_BUF CHANGE OPEN FILE'S BUFFER ADDRESS

FUNCTION This function allows the caller to move an open file's file buffer to another location in memory. Since `READ` and `WRITE` references are by Reference Number, the MLI must memorize the location of the file buffer at `OPEN` time. If the buffer must be moved, this call allows the programmer to inform the MLI and allow it to move the contents of the buffer to the new location. The system memory bit map is updated to reflect the change.

PARAMETER LIST FORMAT

+0	\$02
+1	Reference Number

+2/+3	New Address of File Buffer
-------	----------------------------

REQUIRED INPUTS

+0 Parameter count (2 parameters in list).

+1	Reference number for an open file as returned by OPEN .
----	--

+2/+3 The address (LO/HI) of a new 1024-byte location in which the MLI may maintain the open file's buffer. It must be on an even page boundary (LO byte of address is zero) and not be allocated by the MLI to any open file. The contents of the current file buffer are transferred to this new area, and the old buffer is marked released in the System Global Page memory bit map.

RETURNED VALUES

Return Code	\$000—No errors
-------------	-----------------

\$Ø4 —Parameter count is not \$Ø2

\$43—Invalid reference number

\$56 — Buffer already in use by MLI

\$D3 GET_BUF RETURN OPEN FILE'S BUFFER ADDRESS

FUNCTION	This function returns the address of the file buffer associated with an open file to the caller.
-----------------	--

PARAMETER LIST FORMAT

+0 \$02

+1	Reference Number
----	------------------

+2/+3	Address of File Buffer
-------	------------------------

REQUIRED INPUTS

+0 Parameter count (2 parameters in list).

+1	Reference number for an open file as returned by <code>OPEN</code> .
----	--

RETURNED VALUES

+2/+3 The address (LO/HI) of the 1024-byte file buffer in use by the MLI for this file.

Return Code	\$000—No errors
-------------	-----------------

\$Ø4—Parameter count is not \$Ø2

\$43—Invalid reference number

MLI ERROR CODES

- \$00 No error occurred. Operation completed successfully.
- \$01 Invalid MLI function code number.
- \$04 Incorrect parameter count in parameter list for the function code used.
- \$25 The ProDOS interrupt handler vector table is full. There are already four addresses stored there.
- \$27 A device driver reported an Input/Output error on the media. This could be anything from the diskette drive door being open to a real error on the surface of the diskette.
- \$28 No device is connected for the unit number given. This can happen if no identifiable controller ROM was present in the indicated slot.
- \$2B An attempt was made to write to the disk, but it was write protected. Remove the tape over the write-protect notch if you wish to write on this diskette.
- \$2E In the process of performing an **ONLINE** call, the MLI discovered that a diskette for which there were open files had been removed from its drive and replaced by another volume. Since no check is made when writing to an open file, it is possible that some blocks on the new volume have been damaged.
- \$40 The pathname has invalid syntax. Check to make sure the first byte is a count of the number of characters that follow. Also, be sure that each sub-level index begins with an alphabetic character and that each level is separated from the next by a slash (/).
- \$42 Eight files are open and there is no more room in the MLI's File Control Block (FCB) table for another open file. If you didn't expect any files to be open, set the **LEVEL** to zero and issue a global **CLOSE**.
- \$43 The reference number passed in the parameter list does not denote an open file. Make sure that the **OPEN** call was successful before issuing other calls by reference number.
- \$44 The pathname supplied could not be followed to the final directory. One or more of the subordinate directories in the path did not exist.
- \$45 The volume indicated by the pathname is not currently mounted on any drive.

- \$46 The file indicated by the last name in the pathname was not found in the final directory.
- \$47 A **CREATE** or **RENAME** was attempted and the file named already exists. To perform the operation would create a duplicate entry in the directory.
- \$48 An attempt was made to find one or more free disk blocks (to extend a directory, add a new data block for a file, etc.), but the Volume Bit Map indicates that the diskette is now full.
- \$49 An attempt was made to **CREATE** another file in the Volume Directory, but there are no free entries. Unlike subdirectories, the Volume Directory is of a fixed size (51 entries) and cannot be extended.
- \$4A An earlier version of the ProDOS MLI is being used to read a file which was created with a later version. The older MLI cannot handle this file properly. Use a newer version of ProDOS. This error can also occur if the final subdirectory header has an improper format. The byte at +\$14 in the subdirectory key block (reserved bytes) must contain 5 and only 5 one bits (it is usually \$75).
- \$4B The storage type of a file is not one of the storage types currently supported by this version of ProDOS. Currently, only Seedlings (\$01), Saplings (\$02), Trees (\$03) and Directories (\$0D) are supported.
- \$4C A **READ** operation was attempted and the current file position is at the End of File mark. No data was transferred.
- \$4D An attempt was made to move the file position past the End of File mark. If this position is desired, first move the **EOF** mark.
- \$4E An error occurred having to do with the **ACCESS** bits for a file. Usually this means you attempted to **WRITE** to a write protected file, or you attempted to **DESTROY** or **RENAME** a locked file. You can also get this error if any of the reserved bits are ones for the **ACCESS** byte of a **SET_FILE_INFO** call.
- \$50 An attempt was made to **OPEN**, **RENAME**, or **DESTROY** a previously **OPENed** file. Multiple **OPENs** are only allowed if the file's **WRITE ACCESS** bit is off (write disabled).
- \$51 When searching a directory, it was determined that the count of active file entries in the directory header was larger than the number of entries actually encountered. The directory is damaged and some file entries may be lost.

- \$52 The disk volume which was accessed is not a ProDOS disk. The criteria for determining whether a volume is a ProDOS formatted volume are: the first two bytes of the Volume Directory key block must be zero (previous block pointer); and the byte at offset 4 into the Volume Directory key block must be **\$E** or **\$F** (storage type).
- \$53 One or more of the values in the parameter list is not within its acceptable range. For example, an interrupt handler address of **\$0000** was passed to **ALLOC_INTERRUPT**.
- \$55 At most, only eight “mounted” volumes may be known to ProDOS at one time. Usually this is no problem since only eight files may be open at a time. However, if a single file is open on each of eight different volumes and an **ONLINE** call is made requesting the volume name mounted on a ninth device, this error will result.
- \$56 The address of the I/O file buffer passed to **OPEN** or **SET_BUF** is invalid. The buffer overlaps a previously assigned buffer, memory below **\$200**, or ProDOS itself. The buffer must be in the caller’s memory, and all four of its pages must be marked free in the System Global Page memory bit map.
- \$57 In the process of mounting volumes and recording their names in the Volume Control Block (VCB) table, the MLI discovered two volumes with the same name. Since all file references must be made by volume name and not necessarily by slot and drive, this condition is not permitted.
- \$5A The Volume Bit Map describing the freespace on the volume is damaged. A one bit was found, indicating a free block, for a block outside the legal extent of the volume (for a block number beyond the end of the volume).

PASSING COMMAND LINES TO THE BASIC INTERPRETER

For machine language programs running under the ProDOS **BASIC Interpreter** (BI), an interface is provided to allow execution of command lines created by a program, as if they had been entered from the keyboard. This is the highest level and perhaps the easiest to use ProDOS interface. Through it, a machine language program may easily produce **CATALOG** listings, **DELETE** or **RENAME** files, etc.

To call the BI command handler, place the command string in the monitor **GETLN** line input buffer at **\$200**. The line may be up to 255 characters in

length, and must be followed by a carriage return character (\$8D). The most significant bit of each character should be set, and all alphabetics should be in upper case. Once this has been done, call \$BE03 in the BI's Global Page (JSR \$BE03).

If an error occurs, the BI error code will be placed in the accumulator. Possible codes are listed in Table 6.6.

Table 6.6 BASIC Interpreter Error Codes

Code	Message
\$00	No error
\$01	Not used
\$02	RANGE ERROR
\$03	NO DEVICE CONNECTED
\$04	WRITE PROTECTED
\$05	END OF DATA
\$06	PATH NOT FOUND
\$07	Not used
\$08	I/O ERROR
\$09	DISK FULL
\$0A	FILE LOCKED
\$0B	INVALID PARAMETER
\$0C	NO BUFFERS AVAILABLE
\$0D	FILE TYPE MISMATCH
\$0E	PROGRAM TOO LARGE
\$0F	NOT DIRECT COMMAND
\$10	SYNTAX ERROR
\$11	DIRECTORY FULL
\$12	FILE NOT OPEN
\$13	DUPLICATE FILE NAME
\$14	FILE BUSY
\$15	FILE(S) STILL OPEN

If you wish to print an error message, you need not have a table of messages similar to the above. Instead, place the error number in the A register and call \$BE0C (JSR \$BE0C).

Keep in mind that, unless the machine language program was called by a BASIC program, only direct commands may be issued (as if from the keyboard). BASIC file commands such as OPEN, READ, WRITE, APPEND, and POSITION

will result in a **NOT DIRECT COMMAND** error. Under Apple DOS, commands could be printed with a control-D from an assembly language program, exactly as with BASIC programs. Under ProDOS, this method no longer works. This is because the intercepts used for the “control-D interface” are no longer in the screen output vector, but are now in the Applesoft trace facility, which, of course, isn’t active when your machine language program is running.

COMMON ALGORITHMS

Given below are several pieces of code which may be used when working with ProDOS.

IS PRODOS ACTIVE?

The following series of instructions should be used prior to attempting to call the ProDOS MLI.

```
LDA $BF00      GET MLI VECTOR JMP
CMP #$4C       IS IT A JUMP?
BNE NOPRODS    NO, PRODOS NOT ACTIVE
```

WHAT KIND OF MACHINE IS THIS?

This code will test to determine what type of Apple is running the program.

```

LDA #$08
BIT $BF98      TEST MACHID FROM GLOBAL PAGE
BEQ OLDSYS     OLDER SYSTEM
BPL UNKN       FUTURE SYSTEM - UNKNOWN
BVC APIIC      IT'S AN APPLE IIC
BVS UNKN       OTHERWISE, UNKNOWN
OLDSYS BMI EOR3  EITHER A IIE OR A III
BVC APII       IT'S AN APPLE II
BVS APIIP      IT'S AN APPLE II+
EOR3  BVS APIII IT'S AN APPLE III
...           OTHERWISE IT'S AN APPLE IIE
```

HOW MUCH MEMORY IS IN THIS MACHINE?

This code will determine whether the Apple has 48K, 64K or 128K of RAM.

```
LDA $BF98      GET MACHID FROM GLOBAL PAGE
ASL A          MOVE BITS TO TEST POSITION
ASL A
ASL A
BCC SMLMEM     48K
BMI MEM128     128K
...           OTHERWISE 64K
```

GIVEN A PAGE NUMBER, SEE IF IT IS FREE

This code examines ProDOS's memory bitmap to see if a page is marked free. If so, the page is marked as allocated.

```

BITMAP    EQU $BF58          SEE PAGE 299
          LDA #PAGE          GET PAGE NUMBER (MSB OF ADDR)
          JSR LOCATE          LOCATE ITS BIT IN BITMAP
          AND BITMAP,Y        IS IT ALLOCATED?
          BNE INUSE           YES, CAN'T TOUCH IT
          TXA                 PUT BIT PATTERN IN ACCUM
          ORA BITMAP,Y         MARK THIS PAGE AS IN USE
          STA BITMAP,Y        UPDATE MAP
          ...                 WE'VE GOT IT NOW
LOCATE     PHA                 SAVE PAGE NUMBER
          AND #$07            ISOLATE BIT POSITION
          TAY                 THIS IS INDEX INTO MASK TABLE
          LDX BITMASK,Y        PUT PROPER BIT PATTERN IN X
          PLA                 RESTORE PAGE NUMBER
          LSR A               DIVIDE PAGE BY 8
          LSR A
          LSR A
          TAY                 Y-REG IS OFFSET INTO BITMAP
          TXA                 PUT BIT PATTERN IN ACCUM
          RTS                 DONE
BITMASK    DFB $80,$40,$20,$10    BIT MASK PATTERNS
          DFB $08,$04,$02,$01

```

IS A BASIC PROGRAM RUNNING?

This code will allow your machine language program to determine whether it was called by a BASIC program.

```

LDA $BE42    CHECK BI'S STATE
BEQ NOTRUN   IN IMMEDIATE MODE
...          ELSE, BASIC PROGRAM RUNNING

```

SETTING UP YOUR OWN RESET VECTOR

The code below will set up a user-defined RESET handler.

```

LDA #>RESRTN    SET UP LSB
STA $3F2
LDA #<RESRTN    SET UP MSB
STA $3F3
EOR #$A5        MAKE POWER-UP BYTE
STA $3F4
...             RESET VECTOR READY
RESRTN          RESET HANDLER ROUTINE

```

Note: If you're using the Merlin Assembler, the first line should be LDA #<RESRTN and the third line should be LDA #>RESRTN.

ACTIVATE A PRINTER OR OTHER PERIPHERAL

To activate a printer or other peripheral driver under the ProDOS BASIC Interpreter, do **not** modify the vectors in zero page (CSWL/CSWH or KSWL/KSWH). Doing so will “disconnect” the interpreter and prevent it from intercepting command lines. Instead, store the address of the peripheral driver in BI Global Page in the VECTOUT (\$BE30) or VECTIN (\$BE32) words. The following code will start up a printer in Slot 1.

```

LDA $BE30      SAVE ORIGINAL CONTENTS OF VECTOUT
STA OLDVEC     IN MEMORY SO I CAN TURN THE
LDA $BE31      PRINTER OFF WHEN I'M THRU
STA OLDVEC+1
LDA #$00       PLACE $C100 IN VECTOUT
STA $BE30
LDA #$C1
STA $BE31
...           BEGIN PRINTING VIA COUT
LDA OLDVEC     RESTORE PREVIOUS OUTPUT VECTOR
STA $BE30
LDA OLDVEC+1
STA $BE31

```



Chapter 7

Customizing ProDOS

SYSTEM PROGRAMMING WITH PRODOS

Apple has provided a number of customizing interfaces to ProDOS which allows a programmer to tailor the operation of the system to his specific application needs. These interfaces are considered “safe” and acceptable when working with ProDOS.

Before discussing specific system programming considerations, it is important to understand how ProDOS uses memory and what areas are reserved for its use versus those available for applications programs. Referring to Figure 7.1, the following areas of memory are officially “owned” by the ProDOS Kernel: `$D000-$FFFF` in the language card (primary `$D000-$DFFF` bank); `$BF00-$BFFF`; Zero page locations `$3A-$4F`; and part of the second 4K bank of the language card (starting at `$D100`). The rest of this 4K bank is reserved for the `QUIT` code driver and future uses. The ProDOS Kernel also reserves portions of auxiliary memory (128K) for future use—namely, the same locations it uses in main memory, zero page locations `$80-$FF`, and locations `$200-$3FF`. Apple’s future plans for these memory areas include networking and menu managers, so if you use them you do so at your own risk. In a 128K machine, ProDOS currently sets up an electronic “RAM drive” volume in the auxiliary memory. At present, this volume encompasses most of the auxiliary 64K. In the future, its size may be reduced to accomodate enhancements as mentioned above. You can use auxiliary memory for your own applications if you disable the `/RAM` device driver (see instructions later in this chapter). If the BASIC Interpreter is used, an additional area of memory from `$9600-$BEFF` is allocated to its use. `$3D0-$3FF` is used as a system vector area as defined by the *Apple II Reference Manual for the IIe Only*.

Note that ProDOS routines, including the clock driver, make heavy use of `$200-$2FF`, the monitor `GETLN` input line buffer. If your programs use this area, you should not depend upon it across ProDOS system calls. You should also be aware of the fact that the MLI cannot be called from memory in the

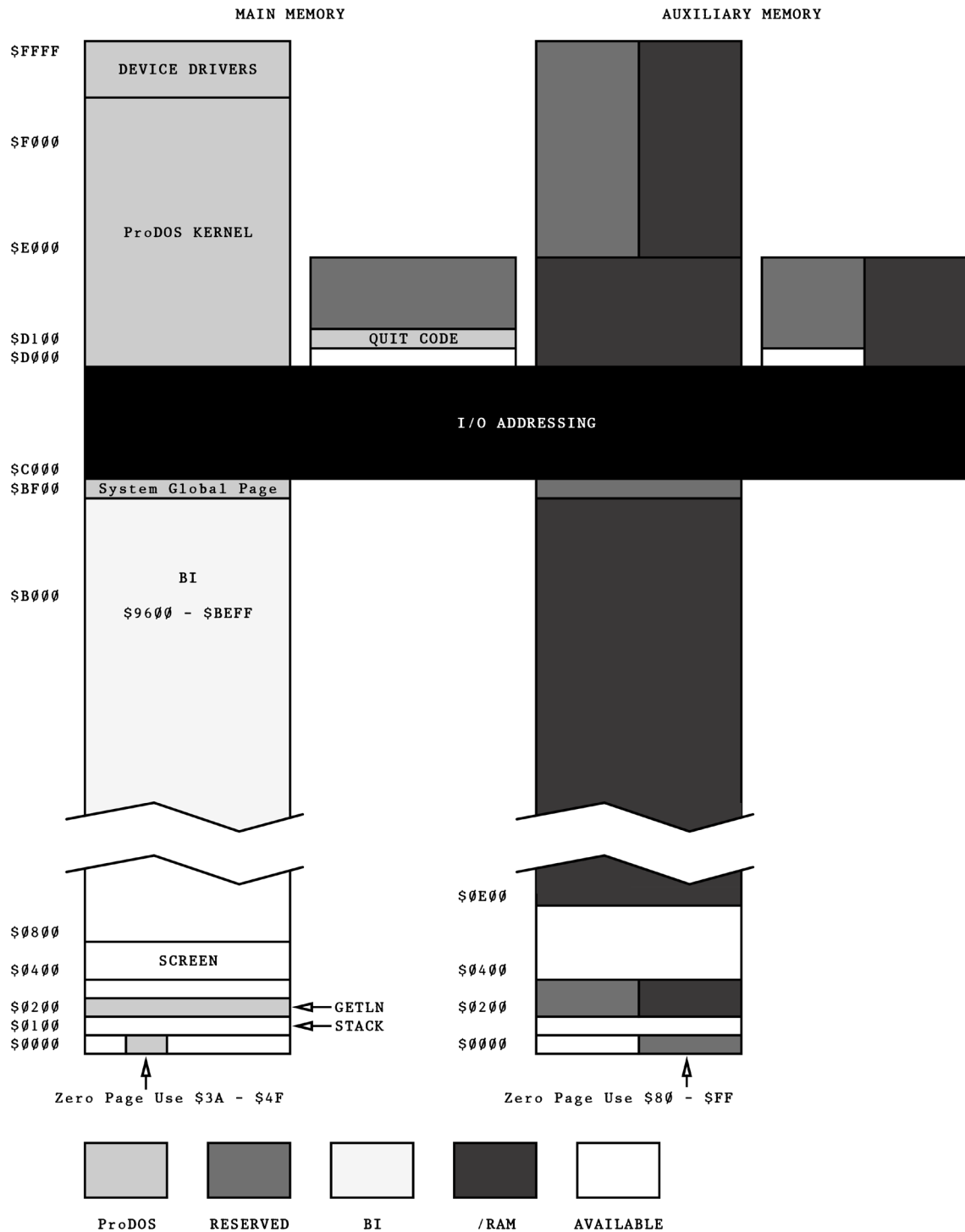


Figure 7.1 ProDOS Memory Usage

auxiliary bank, and that memory outside the area between \$2000 and \$BEFF in the main RAM bank may not be used for buffers passed to the MLI.

INSTALLING A PROGRAM BETWEEN THE BI AND ITS BUFFERS

Once in a while it is useful to find a “safe” place in memory to put a machine language program (a printer driver, or external command handler, perhaps) where BASIC and ProDOS will never walk over it. If the program is less than 200 bytes long, \$3000 is a good choice. For larger programs, it is usually better to “tuck” the program in between the ProDOS BASIC Interpreter and its file I/O buffers. The program need not be relocatable, since the BI will always be in the same place in memory, and the program can be placed at a fixed location just below it (see Figure 5.1). More than one program may be “tucked” in this area, but this may require one or more of them to be relocated, depending upon the order in which they are loaded.

To request space for a program, you must execute a call to the BI’s buffer allocation subroutine using a vector in the BI Global Page. You may request a buffer of any size as long as it is an even multiple of pages (one page is 256 bytes). When called, the buffer allocation routine relocates any open file buffers as well as its General Purpose Buffer downward in memory, lowering Applesoft’s HIMEM pointer as necessary, and returns the address of the first page in the new buffer. The new buffer will be placed directly below \$9A00. Subsequent calls to the buffer allocation routine will cause allocations of buffers below earlier ones. The BI file buffers will always be lower in memory than any externally allocated buffers. When you are finished with all of the buffers you have allocated, you may free all of them with a single call. There is no provision for freeing individual buffers.

To allocate a buffer, invoke the following subroutine:

GBUFF	LDA #4	ALLOCATE 4 PAGES (1024 BYTES)
	JSR \$BEF5	CALL GETBUFR
	BCS ERROR	DID AN ERROR OCCUR?
	STA BUFMSB	STORE BUFFER ADDRESS MSB
	LDA #0	
	STA BUFLSB	STORE BUFFER ADDRESS LSB
	RTS	ALL DONE

To free all buffers you have allocated:

```
FBUFS JSR $BEF8 CALL FREEBUFR
```

Note that you may allocate as many buffers as you wish using the **GBUFF** subroutine, but that a single call to **FBUFFS** frees all buffers.

ADDING COMMANDS TO THE PRODOS BASIC INTERPRETER

There exists a well defined interface to allow you to write your own command handlers for the ProDOS BASIC Interpreter. Suppose, for example, that you wish to add a **COPY** command which will accept an input pathname, followed by a comma and an output pathname. You can write a handler for such a command in assembly language, install the handler between the BI and its buffers (see the previous section), and then inform the BI of its existence. Every time the BI receives a command line it doesn't recognize, it will pass it through to your handler before passing it to Applesoft. Note that this implies that your command's name must be different from any existing ProDOS command name. You may not replace or supercede an existing ProDOS command.

To install your own command handler, place its entry point address in the vector in the BI Global Page at **\$BE07** and **\$BE08**. These two bytes are the address portion of a Jump (**JMP**) instruction (**EXTERNCMD**) which normally points to a Return from Subroutine (**RTS**) instruction within the BI. It is not a good idea to assume that this address is pointing to an **RTS** since someone else's command handler could have been previously installed. To make sure you do not "disconnect" an earlier installed command handler and yours is "daisy chained" to it, save the address you find in **EXTERNCMD + 1** and branch to it from your handler if the command line passed is not your command.

Each time the BI scans a command line and cannot find the command name in its table of valid names, it will call your routine. Your program should compare the command in the command line with yours. The address of the command line is in **VPATH1** (**\$BE6C/\$BE6D**) in the BI Global Page. The command line consists of a length byte followed by one or more ASCII characters with their most significant bit off. If the command is not yours, jump to the next handler (previous contents of **\$BE07/\$BE08**) with the carry set (**SEC**) to indicate the command is not yours. If the command is yours, there are two options. If the command's syntax is not compatible with other ProDOS commands (i.e. it has non-standard operands or keywords), you may immediately begin performing the function indicated. When the program finishes, it should store a zero in **PBITS** in the BI Global Page (**\$BE54**) to indicate no operands are to be parsed, and return (**RTS**) with the carry clear (**CLC**). In this case, do

not **JMP** to the next handler as you would if the command was not yours. If, on the other hand, the command has standard ProDOS syntax, you can use the BI's syntax scanner to pick off the operands and optional keywords. To do this, once you have identified the command as yours, store the address of the beginning of your code which will process the command (after the syntax scan) in **XTERNADDR** (\$BE50/\$BE51) in the BI Global Page, store a \$00 in **XCNUM** (\$BE53) to indicate that this is an external command, and store the length of your command name (less one) in **XLEN** (\$BE52) so that the BI will know where to start looking for operands. You should also set up **PBITS** (two bytes of flags) in the BI Global Page to describe the operands the BI is likely to find on your command. If you have a very simple command with only a pathname as an operand, you can set **PBITS** to \$01, \$00. If you want the BI to automatically provide the prefix of the current volume (default slot, drive) as well as allow the S and D keywords, set **PBITS** to \$01, \$04. Once you have set up **XTERNADDR**, **XCNUM**, **XLEN**, and **PBITS**, return (**RTS**) to the BI with the carry clear (**CLC**). When the command line has been successfully scanned, control will return to your handler at the location you indicated in **XTERNADDR**. If a **SYNTAX ERROR** occurs, control will not return. When your command handler completes its tasks, it may return to the BI with an **RTS** instruction (the carry here is insignificant). Your handler need not save or restore any registers.

An example of a command handler is given in Appendix A. This program installs a handler between the BI and its buffers, and connects it to ProDOS through the **EXTERNCMD** vector. If the ProDOS user enters the command “**TYPE**” followed by a pathname, the command handler reads the indicated file and prints it on the screen.

DISABLE /RAM VOLUME FOR 128K MACHINES

If your application needs to use the additional 64K in the Extended 80-column Card (or the alternate 64K bank in the IIc) for its own purposes, rather than as an electronic disk drive (RAM drive), you should disable the **/RAM** device driver. You might want to do this if you plan to use the “double HIRES” graphics feature of the Apple IIe and IIc, for example.

The **/RAM** device driver is installed by the ProDOS Loader/Relocator when the Kernel is loaded. Part of it resides in the Kernel itself (from \$FF00-\$FF7F), and the remainder remains in auxiliary memory at \$200-\$3FF. Its address is placed in the list of device drivers for Slot 3, Drive 2 in the System Global Page.

One way to avoid conflicts between the /RAM and your application is the BSAVE a dummy file such that its blocks will coincide with the area of memory you will be using. If you BSAVE an 8K file to /RAM (before any other operations on the /RAM volume), it will fall across \$2000-\$3FFF, the primary HIRES buffer. If you save a second 8K file it will fall across \$4000-\$5FFF, the secondary HIRES buffer. This is the easiest way to use “double HIRES” graphics while leaving the /RAM volume partially available for your use as an electronic disk drive.

If you want to totally disable the /RAM device driver, you must remove its entry from the System Global Page device driver vector list (DEVADR32). You must also remove the device number for Slot 3, Drive 2 from the online devices list (DEVLST), and reduce the device count (DEVCNT) by one. If you plan to reinstall the /RAM volume later, be sure to save the contents of DEVADR32 in a safe place so you can later restore it. Note that it is good programming practice to leave /RAM installed upon exiting your program so that other applications may use it. Reinstalling /RAM erases (“formats”) the volume, so you should not reinstall it upon entry to an application which will be reading files passed via the /RAM volume by a previous application.

The following subroutine will remove the /RAM driver, allowing alternate uses of the auxiliary 64K:

```

      2  * START BY CHECKING TO SEE IF /RAM COULD BE THERE
0000: AD 98 BF 4  REMOVE LDA $BF98 CHECK MACHID
0003: 29 30 5  AND #$30 ISOLATE MEMORY BITS
0005: C9 30 6  CMP #$30 128K?
0007: D0 10 7  BNE NORAM NO - NO AUX MEMORY
0009: AD 26 BF 8  LDA $BF26
000C: CD 16 BF 9  CMP $BF16 IF SLOT 3, DRV <> DRV 2 VECTOR..
000F: D0 0A 10 BNE GOTRAM THEN IT'S INSTALLED
0011: AD 27 BF 11 LDA $BF27
0014: CD 17 BF 12 CMP $BF17
0017: D0 02 13 BNE GOTRAM
0019: 38 14 NORAM SEC ;
001A: 60 15 OKXIT RTS
      17 * SAVE OLD VECTOR AND REMOVE IT
001B: AD 26 BF 19 GOTRAM LDA $BF26 SAVE OLD VECTOR CONTENTS
001E: 85 59 20 STA OLDVEC
0020: AD 27 BF 21 LDA $BF27
0023: 85 5A 22 STA OLDVEC+1
0025: AD 16 BF 23 LDA $BF16 POINT IT AT NISTALLED DEV
0028: 8D 26 BF 24 STA $BF26
002B: AD 17 BF 25 LDA $BF17
002E: 8D 27 BF 26 STA $BF27
      28 * SQUISH OUT DEVICE NUMBER FROM DEVLST
0031: AE 31 BF 30 LDX $BF31 GET DEVCNT
0034: BD 32 BF 31 DEVLDP LDA $BF32,X PICK UP LAST DEVICE NUM
0037: 29 70 32 AND #$70 ISOLATE SLOT
0039: C9 30 33 CMP #$30 SLOT = 3?
003B: F0 05 34 BEQ GOTSLT YES, CONTINUE
003D: CA 35 DEX
003E: 10 F4 36 BPL DEVLDP CONTINUE SEARCH BACKWARDS
0040: 30 D7 37 BMI NORAM CAN'T FIND IT IN DEVLST
0042: BD 33 BF 38 GOTSLT LDA $BF32+1,X GET NEXT NUMBER

```

```

0045: 9D 32 BF 39          STA  $BF32,X    AND MOVE THEM FORWARD
0048: E8                40          INX
0049: EC 31 BF 41          CPX  $BF31    REACHED LAST ENTRY?
004C: D0 F4                42          BNE  GOTSILT  NO, LOOP
004E: CE 31 BF 43          DEC  $BF31    REDUCE DEVCNT BY 1
0051: A9 00                44          LDA  #$00    ZERO LAST ENTRY IN TABLE
0053: 9D 32 BF 45          STA  $BF32,X
0056: 18                46          CLC
0057: 90 C1                47          BCC  OKXIT   BRANCH ALWAYS
0059: 00 00                49  OLDVEC  DW      0    OLD VECTOR SAVE AREA

```

To reinstall the /RAM driver, execute this subroutine:

```

51  * SEE IF SLOT 3 HAS A DRIVER ALREADY
53  HIMEM  EQU  $73    PTR TO BI'S GENERAL PURPOSE BUFFER
005B: AE 31 BF 55  INSTALL  LDX  $BF31    GET DEVCNT
005E: BD 32 BF 56  INSLP   LDA  $BF32,X  GET A DEVNUM
0061: 29 70 57          AND  #$70    ISOLATE SLOT
0063: C9 30 58          CMP  #$30    SLOT 3?
0065: F0 3F 59          BEQ  INSOUT   YES, SKIP IT
0067: CA        60          DEX
0068: 10 F4 61          BPL  INSLP    KEEP UP THE SEARCH
63  * RESTORE THE DEVNUM TO THE LIST
64          LDX  $BF31    GET DEVCNT AGAIN
006A: AE 31 BF 64          CPX  $0D    DEVICE TABLE FULL
006D: E4 0D 65          BNE  INSLP2
006F: D0 00 66
67  * ERROR ... YOUR ERROR ROUTINE
68  INSLP2  LDA  $BF32-1,X  MOVE ALL ENTRIES DOWN
0074: 9D 32 BF 69          STA  $BF32,X  TO MAKE ROOM AT FRONT
0077: CA        70          DEX
0078: D0 F7 71          BNE  INSLP2
007A: A9 B0 72          LDA  #$B0
007C: 8D 32 BF 73          STA  $BF32    SLOT 3, DRIVE 2 AT TOP OF LIST
007F: EE 31 BF 74          INC  $BF31    UPDATE DEVCNT
76  * NOW PUT BACK THE DEVICE DRIVER VECTOR
78          LDA  OLDVEC
0084: 8D 26 BF 79          STA  $BF26
0087: A5 5A 80          LDA  OLDVEC+1
0089: 8D 27 BF 81          STA  $BF27
83  * FINALLY, REFORMAT THE /RAM VOLUME
85          LDA  $BF32
008C: AD 32 BF 85          STA  $43    DEVNUM = SLOT 3, DRIVE 2
008F: 85 43 86          LDA  #3
0091: A9 03 87          STA  $42    CMD = FORMAT
0093: 85 42 88          LDA  HIMEM    512-BYTE BLOCK BUFFER
0095: A5 73 89          STA  $44    (PAGE ALIGNED)
0097: 85 44 90          LDA  HIMEM+1  WE CAN USE BI'S G.P. BUFFER
0099: A5 74 91          STA  $45    (IF BI IS AROUND)
009B: 85 45 92          STA  $C080   SELECT L.C. FOR DRIVER
009D: 8D 80 C0 93          JSR  RAMDRV  GO FORMAT THE VOLUME
00A0: 20 A7 00 94          STA  $C081   SELECT MOTHERBOARD ROMS
00A3: 8D 81 C0 95          INROUT  RTS    ; AND EXIT TO CALLER
00A6: 60 96          RAMDRV  JMP  ($BF26) <<< JUMP TO /RAM DRIVER >>>
00A7: 6C 26 BF 97

```

WRITING YOUR OWN INTERPRETER

A ProDOS “Interpreter” (also known as a “System Program”) is a machine language program which stands between the user and the ProDOS MLI, providing a function. An interpreter may be executed by the smart RUN

command (“-”), may be invoked at boot time, or may be executed upon leaving another ProDOS interpreter. Interpreters are stored in **SYS** files on a ProDOS volume, and are initially loaded at **\$2000**, although they may include code to relocate themselves elsewhere once they begin execution. Examples of interpreters are **BASIC.SYSTEM** (the “BI”), **FILER**, **CONVERT**, and **EDASM.SYSTEM**. According to convention, an interpreter must be able to pass control to any other interpreter when it exits.

When writing your own interpreter, you must be aware of these considerations:

1. You must **BSAVE** your interpreter as a “**SYS**” type file from location **\$2000**. If you want to include code to execute elsewhere in the machine, you may include a front-end which relocates the rest of the program (this is what the BI does). Normally, the memory available to you in a 64K system includes **\$800-\$BEFF**. If you are running in a 48K machine, the ProDOS Kernel occupies memory from **\$9000-\$BFFF** so you are limited to **\$800-\$8FFF** for your program.
2. If you want your interpreter to be automatically executed as the first interpreter when ProDOS boots, you must name it “**xxxx.SYSTEM**”, here **xxxx** can be any name. It must also be the first **SYS** file using that naming convention to be found in the Volume Directory of the boot diskette.
3. In order to insure correct operation of the interrupt handler in the ProDOS Kernel, set the stack register (**S**) to point to the top of the stack page (**\$FF**) upon entry, and do not use more than the top three quarters of the stack. The interrupt handler assumes that the last item on your stack is stored at **\$1FF**, when it makes its determination of whether or not to save part of the contents of the stack before invoking an interrupt driver routine.
4. As soon as your program begins execution, it should set up the **POWERUP** byte in page 3 and three areas in the System Global Page as follows.

\$03F4: POWERUP byte

\$BF58: BITMAP (system memory bit map)

\$BFFC: IBAKVER (minimum version of MLI acceptable)

\$BFFD: IVERSION (version number of your interpreter)

When your interpreter gets control, it should first set up the **RESET** vector at **\$3F2/\$3F3** to point to its own **RESET** handler and fix the **POWERUP** byte at **\$3F4** accordingly. The **POWERUP** byte should be fixed even if you do not replace the **RESET** handler address (unless you want to reboot on **RESET**). To fix the **POWERUP** byte, exclusive **OR** the contents of **\$3F3** with **#A5** and store the result in **\$3F4**.

A subroutine for checking the system memory bit map was given in Chapter 6. Use this to mark those areas in memory which your program will use. Do not mark areas which may be used for MLI buffers. By doing this, the MLI can keep a watchful eye on the execution of your program to prevent accidental overlay of your code with buffers. To determine what values to use for **IBAKVER** and **IVERSION**, examine memory in the version of ProDOS you are using for development and note the values at **\$BFFE** (**KBAKVER**) and **\$BFFF** (**KVERSION**). Assemble the values you find there as constants into your program, and use these to initialize **IBAKVER** and **IVERSION**.

5. If you wish to use 80 columns, first check the **MACHID** byte in the System Global Page to see if 80 columns are available and then call (**JSR**) **\$C300**. To disable 80-column hardware, load a **#15** into the A register and call **\$C300**. Avoid using the Apple IIe and IIc 80-column soft switches, because these will not work for third party 80-column cards or in an Apple II or Apple II Plus.
6. When your program is ready to exit, close all open files, reinstall the **/RAM** driver if you disconnected it previously, and execute the following code.

EXIT	DEC \$3F4	FORCE REBOOT ON RESET
	JSR \$BF00	CALL THE MLI
	DFB \$65	QUIT CALL
	DW PARMS	
PARMS	DFB 4	4 PARMS
	DFB 0	QUIT TYPE = 0
	DW 0	RESERVED
	DFB 0	RESERVED
	DW 0	RESERVED

The MLI will free any memory you have allocated in the system bit map. It will then prompt the user for a new prefix and pathname for the next interpreter, and will load it and execute it. The code which performs these tasks is at **\$D100-\$D3FF** in the secondary 4K block of the language card. It is moved by the MLI to **\$1000-\$12FF** before execution. You may create your own quit code by replacing the three pages of code image in the language card if you wish.

INSTALLING NEW PERIPHERAL DRIVERS

If you are writing for a peripheral, such as a printer or disk drive, you should be aware of the conventions to which ProDOS adheres when examining and calling drivers.

If your driver is in ROM on the peripheral card itself, it should follow the Apple II standards for peripherals as follows.

For Non-Disk Devices

Address	Value
\$Cs05	\$38 (standard BI requirement)
\$Cs07	\$18 (standard BI requirement)
\$Cs0B	\$01 (generic signature of firmware cards)
\$Cs0C	\$ci (specific device signature)

The device signature is made up of two nibbles. “c” defines the class of the device as shown below. The second nibble, “i”, is a specific device identifier assigned by Apple Computer, Inc.

“c” Nibble	Class
\$0	reserved
\$1	printer
\$2	joystick or X-Y input device
\$3	serial or parallel card
\$4	modem
\$5	sound or speech device
\$6	clock
\$7	mass storage device
\$8	80-column card
\$9	network or bus interface
\$A	special purpose (other)
\$B-\$F	reserved

ProDOS makes the following special check for a clock:

Address	Value
\$Cs00	\$08 (unique device signature for the Thunderclock)
\$Cs02	\$28
\$Cs04	\$58
\$Cs06	\$70

For Disk Devices

Address	Value
\$Cs01	\$20 (unique disk device signature)
\$Cs03	\$00
\$Cs05	\$03
\$Cs07	\$3C
\$CsFC/D	Disk capacity in blocks (non-DISK II)
\$CsFE	Status bits (non-DISK II)
	1... removable media
	.1.. interruptable device
	..nn number of volumes on device

```

.... 1... format allowed
.... .1.. write allowed
.... ..1. read allowed
.... ...1 status read allows
ProFILE status bits = $47
$CsFF $00 = DISK II
$xx = LSB of Block device driver in ROM
      for non-DISK II ($Csxx).
      ProFILE hard disk $xx = $EA
      $xx may not equal $FF

```

If your device driver is in RAM (below `$C000`), and you are invoking it using the BASIC Interpreter's commands `PR# A$xxxx` or `IN# A$xxxx`, the first byte of your code must be a `CLD` instruction (`$D8`); otherwise, the BI will not recognize your routine as a valid driver. If your routine is short, you can place it in the `$300-$3EF` range. If it is longer, you can call the BI's buffer allocation routine (previously covered in this chapter) to place it between the BI and its buffers.

INSTALLING AN INTERRUPT HANDLER

If you plan to use a peripheral card which supports interrupts, you may want to write an interrupt handler for that card. You should use the ProDOS first level interrupt handler in the Kernel so that other cards may also service their interrupts. To do this, use the `MLI:ALLOC_INTERRUPT` call to install your interrupt handler's entry address in the interrupt vector table within the ProDOS System Global Page. When writing an interrupt handler, follow these steps in the order indicated.

1. Make sure your interrupt handler is stored in main memory between `$200` and `$BEFF`.
2. Call the `MLI` with the `ALLOC_INTERRUPT ($40)` call to cause your routine's entry point to be placed in the vector table.
3. Perform whatever I/O is necessary specific to your peripheral to enable its interrupt generating mechanism.

When your interrupt routine is called, the first instruction executed should be a `CLD` (to let ProDOS know that this is a valid externally written routine). You should then determine whether the interrupt which caused your routine to be invoked was indeed from your peripheral. If it was not, return to the Kernel with the carry flag set. If it was, service the interrupt, and upon completion,

return to the Kernel with the carry flag clear. Your interrupt handler need not save or restore any registers, and it may use up to 16 bytes of stack space and zero page locations **\$FA** through **\$FF** (these are saved and restored by the Kernel). The Kernel assumes that the “bottom” of the stack is at **\$1FF** when it determines what to save. Your application should always start the stack pointer at **\$FF**. Note that the motherboard ROM is deactivated in an interrupt handler routine (do not attempt to print via **\$FDED**, for example).

If you wish to remove a previously installed interrupt routine, first disable the interrupt generation mechanism on your peripheral card to prevent further interrupts from occurring, then call the **MLI:DEALLOC_INTERRUPT** function to remove your handler from the list.

When writing an interrupt service routine, you should minimize the actual function performed “on the interrupt.” If you are collecting data from a serial port which will later be written to disk, do not write the data while in the interrupt service routine, since this may adversely impact the performance of the program which was executing when the interrupt occurred, or it may cause you to “lose” subsequent interrupts while processing the first. Instead, use the interrupt routine to fill a “circular buffer” which is periodically dumped to disk by the interrupted program. An example of this technique and of writing interrupt handlers in general is given in the **DUMBTERRM** program in Appendix A.

If you wish to call the MLI while in an interrupt routine, you should take steps to allow any interrupted MLI call to complete before using the MLI yourself (the MLI is not reentrant). Check the **MLIACTV** flag (**\$BF9B**) in the System Global Page to see if the MLI is active. If the MLI is not active, you may issue MLI calls immediately. If the MLI is active, save the contents of **CMDADR** (**\$BF9C**) and replace it with an address within your service routine. Then return to the Kernel with the carry clear. When the MLI call completes, control will be passed to you instead of the original MLI caller. You should carefully save all registers, perform your processing, as needed, restore the registers again, and jump to the saved contents of **CMDADR** to allow the original caller to continue. Note that you can be interrupted during your processing unless you disable interrupts. If you are not careful, a subsequent interrupt could cause your interrupt service routine to overwrite the saved contents of **CMDADR** with an address within your own program, causing an infinite loop! It might be a good idea to set a flag when saving **CMDADR** and clear it only when you have completed all processing. Your interrupt service routine can then check the flag and discard any interrupts which occur while you are finishing up processing of the first interrupt.

DIRECT MODIFICATION OF PRODOS — A WORD OF WARNING

Making changes to your copy of ProDOS should only be undertaken when absolutely necessary. In the past, many third party software packages were sold for DOS, the earlier Apple II operating system, which patched or made wholesale changes. Because of the dependency these programs had on fixed locations within DOS and their importance to the collective software offering for the machine, programmers at Apple felt hampered in their efforts to improve DOS. Bugs in DOS could only be fixed with patches to existing code—no reassembly could be performed on the DOS code as this would cause critical locations to move “out from under” existing applications. With the introduction of ProDOS, Apple started out fresh. Earlier shortcomings in DOS have been corrected with ProDOS and numerous enhancements have been added. Hopefully, most packages written for ProDOS will not have to depend on changing the operating system’s code itself. In any case, be forewarned: Apple will not hesitate to reassemble the ProDOS Kernel or the BASIC Interpreter or other ProDOS components if changes are desirable, and the stated policy is that programs which depend on locations or entry points which are not published by Apple will do so at their own risk.

Although ProDOS provides most of the functionality needed by the BASIC or assembly language programmer, at times a custom change is desirable. When making a change, weigh its value against the difficulty of reconstructing and reapplying it for later versions of ProDOS as they become available. Of course, if you never plan to upgrade your version of ProDOS this is not a concern. In addition, wholesale modification of ProDOS without a clear understanding of the full implications of each change can result in an unreliable system.

APPLYING PATCHES TO PRODOS

The usual procedure for making changes to ProDOS involves “patching” the object or machine language code in ProDOS. Once a desired change is identified, a few instructions are stored over other instructions within ProDOS to modify the program. There are three levels at which changes to ProDOS may be applied.

- New code may be written and added to ProDOS through a “standard” interface. If this is done, as in the case of an interrupt handler, for example,

there need not be any ProDOS version dependencies involved. Examples of this type of modification have been given earlier in this chapter.

- A patch may be applied to a ProDOS system component, such as the Kernel or the BASIC Interpreter, directly in memory. If this is done, a later reboot will cause the change to “fall out” or be removed. This method is usually used to test a change before making it permanent.
- A patch may be made directly to the diskette containing the ProDOS system component in question. Most ProDOS components are stored as **SYS** files and may be **BLOAd**ed, modified using the monitor, and **BSAVe**d back to diskette. If a change is to be made to the bootstrap loader (stored in block 0 of the volume), a sector edit/or or the **ZAP** program given in Appendix A must be used. When applying patches to the **BASIC.SYSTEM** or **PRODOS** files, you can find a location within the unrelocated image of the BI or the Kernel if you know its address in the relocated and running version. To do this, refer to Table 7.1. For example, if you wish to patch \$9B7C in the BI, you must patch \$257C after **BLOAD**ing **BASIC.SYSTEM**. If you wish to change \$D32A in the MLI, **BLOAD PRODOS** and change \$302A.

Table 7.1a ProDOS Patch Locations for FILE = “PRODOS” (64K)

EXECUTION ADDRESS	IMAGE ADDRESS	EXECUTION ADDRESS	IMAGE ADDRESS
BF00	4E00 (64K system global page image)	E800	4500
D000	2D00 (alternate 4K: 5900—QUIT Code)	E900	4600
D100	2E00 (alternate 4K: 5A00)	EA00	4700
D200	2F00 (alternate 4K: 5B00)	EB00	4800
D300	3000	EC00	4900
D400	3100	ED00	4A00
D500	3200	EE00	4B00
D600	3300	EF00	4C00
D700	3400	F000	4D00
D800	3500	F100	zeroed (clock code from F142 from 5000)
D900	3600	F200	zeroed
DA00	3700	F300	zeroed
DB00	3800	F400	zeroed
DC00	3900	F500	zeroed
DD00	3A00	F600	zeroed
DE00	3B00	F700	zeroed
DF00	3C00	F800	5200 (diskette driver)
E000	3D00	F900	5300
E100	3E00	FA00	5400
E200	3F00	FB00	5500

EXECUTION ADDRESS	IMAGE ADDRESS	EXECUTION ADDRESS	IMAGE ADDRESS
E300	4000	FC00	5600
E400	4100	FD00	5700
E500	4200	FE00	5800
E600	4300	FF00	2B00 (/RAM device driver through FF99)
E700	4400	FF80	5080 (Interrupt vectors and handler)

Table 7.1b ProDOS Patch Locations for FILE = "BASIC.SYSTEM"

EXECUTION ADDRESS	IMAGE ADDRESS	EXECUTION ADDRESS	IMAGE ADDRESS
9A00	2400 (BI image)	AC00	3600
9B00	2500	AD00	3700
9C00	2600	AE00	3800
9D00	2700	AF00	3900
9E00	2800	B000	3A00
9F00	2900	B100	3B00
A000	2A00	B200	3C00
A100	2B00	B300	3D00
A200	2C00	B400	3E00
A300	2D00	B500	3F00
A400	2E00	B600	4000
A500	2F00	B700	4100
A600	3000	B800	4200
A700	3100	B900	4300
A800	3200	BA00	4400
A900	3300	BB00	4500
AA00	3400	BC00	4600
AB00	3500	BE00	4700 (BI Global Page image)

The patches given here are applied directly to a diskette with ProDOS Version 1.0.1 (1 January 1984). You must reboot after making any changes in order to cause them to take effect. Do not make these changes to your original ProDOS System diskette. Modify a copy so you can "back out" any changes you make by copying the original again.

CHANGING THE NAME OF THE STARTUP FILE

You can change the name of the **STARTUP** file which the BI executes at bootup by patching the first block of **BASIC.SYSTEM** as follows.

```
BLOAD BASIC.SYSTEM,TSYS,A$2000  
CALL -151  
21E5:05 48 45 4C 4C 4F  
BSAVE BASIC.SYSTEM,TSYS,A$2000
```

Here we are changing the name from **STARTUP** to **HELLO**. The first byte indicates the number of characters in the name (5) and may be a maximum of 7 characters. Each ASCII byte should have its most significant bit off. The Startup file may be of any type which can be run using the “-” (Smart **RUN**) command.

PUT THE CURSOR ON COMMAND THAT CAUSED PRODOS ERROR

When you get a ProDOS error message such as “**PATH NOT FOUND**” or “**FILE TYPE MISMATCH**” because you typed the wrong file name or misspelled it slightly, it would be nice if ProDOS would return the cursor on the line with your faulty command so you could easily retype it. To make ProDOS do this from now on, apply the following patches.

```
BLOAD BASIC.SYSTEM,TSYS,A$2000  
CALL -151  
257C:4C C0 BB  
45C0:A4 25 88 88 88 84 25 20 22 FC 4C 3F D4  
BSAVE BASIC.SYSTEM,TSYS,A$2000
```

NOTE: The patches described on this page are for Version 1.0.1 of ProDOS and probably will not work with other versions.

HOW TO WRITE TO A DIRECTORY FILE

The ProDOS MLI will not allow explicit **WRITEs** to a directory file under any circumstances (it makes no difference whether the **DIR** file is “locked” or not). Under normal conditions, the only program which may modify a directory file is the MLI itself (when **CREATE**ing a new file, updating the info in an old one, or **DESTROY**ing one). If you wish to directly modify a directory entry with your own program, you should follow this procedure to circumvent the MLI.

1. Open the directory file using **MLI:OPEN**.
2. **READ** the block requiring update.

3. Execute the following code to find the block number.

```

LDA $C08B          SELECT RAM CARD
LDA $C08B
LDA REFNUM          PICK UP REF NUM OF FILE
CLC
SBC #0              MAKE IT AN OFFSET
LSR A               *32 FOR INDEX INTO FCB'S
ROR A
ROR A
ROR A
TAX
LDA $F310,X          GET CURRENT BLOCK NO.
STA BLKNUM
LDA $F311,X
STA BLKNUM+1
STA $C081            SELECT MOTHERBOARD ROMS

```

4. Use `MLI:WRITE_BLOCK` to write back the block. Note that `$F310` and `$F311` may be version-dependent locations.

CREATING A NEW FILE TYPE

When you **CREATE** a file with the MLI, you may specify any file type you wish. If you wish to define a new file type for your application, pick a number between `$F1` and `$F8`. When a **CAT** or **CATALOG** command is issued in the BASIC Interpreter, the file type listed will be a “`$Fn`”. If you want to use a three letter abbreviation instead, you must modify the table in the BI. The



A rare moment when a new ProDOS file type is born...

patch given below is highly version dependent and will only work for ProDOS Version 1.0.1 (1 January 1984).

The first thing to do is examine the table of file types in the BI at **\$B9DB**. This table consists of 14 entries of one byte each, giving the ProDOS file type number for each of the supported types. You will have to replace one of the entries that you never use with your own file type. The entries need not be in numerical order. Immediately following the type table is a table of 3-byte entries giving the names which correspond to the numeric types. This table is in reverse order to the first and begins at **\$B9E9**. As an example, suppose you wished to replace the last entry in the tables, **\$19** “ADB”, with **\$F1** “ABC”.

```
BLOAD BASIC.SYSTEM,TSYS,A$2000  
CALL -151  
43E8:F1  
43E9:C1 C2 C3  
BSAVE BASIC.SYSTEM,TSYS,A$2000
```

Notice that **\$B9E8** maps to **\$43E8** in the unrellocated image of **BASIC.SYSTEM**.

Note: The patches described above are for version 1.0.1 of ProDOS and probably will not work with other versions.

RECOVERING DATA FROM A DAMAGED DISK

If one of the sectors which makes up a block is damaged, ProDOS will return with an I/O error. If in fact the error was in the second half of the block, the first half will be read into memory before the I/O error occurs. However, if the error is in the first half of the block, ProDOS will not attempt to read the second half. To recover the second, undamaged sector of the block, the following patch will force ProDOS to ignore any errors while reading the first half of a block. Errors while reading the second half will still behave normally.

```
BLOAD PRODOS,TSYS,A$2000  
CALL -151  
5228:00  
BSAVE PRODOS,TSYS,A$2000
```

The above patch, while it will work properly with undamaged blocks, is not advisable in normal use as it will fail to indicate when errors have occurred.

USING PRODOS WITH 40-TRACK DRIVES

The device driver supplied with ProDOS supports only 35 tracks. The code can be modified easily to support third party disk drives with 40 tracks, but there

are a couple of things to consider. The patch will apply to all drives (regardless of the number of tracks supported) connected to Disk II or compatible controller cards. This should cause no difficulties even if one 35-track and one 40-track drive are on the same controller card. Because you will also want to format 40-track disks, it will be necessary to modify **FILER**. The patch to **FILER** will apply to all disks that you format, and will produce an error if you attempt to format a disk on a drive supporting less than 40 tracks.

This patch modifies the Disk II Driver, which is a part of the “**PRODOS**” file (or “**P8**” file), so that it allows 320 blocks per volume instead of 280 blocks per volume. First set the prefix to the directory that contains the file you want to modify. This file will normally be called “**PRODOS**” on an 8-bit Apple II and “**P8**” on a 16-bit Apple IIGS. If the file name is not “**PRODOS**,” substitute the correct filename wherever “**PRODOS**” appears.

```
UNLOCK PRODOS
BLOAD PRODOS,TSYS,A$2000
CALL -151
address*:40
3D0G
BSAVE PRODOS,TSYS,A$2000
LOCK PRODOS
```

* “address” varies with the version of ProDOS, as follows:

ProDOS Version	address
1.0.1	520D
1.0.2	52CD
1.1.1	56E3
1.2	58E3
1.3	58E3

The following patch modifies the program **FILER*** to format 40 tracks instead of 35. After this modification is made, only 40-track drives may be formatted with **FILER**.

```
UNLOCK FILER
BLOAD FILER,TSYS,A$2000
CALL -151
addr**:40
79F4:28
3D0G
BSAVE FILER,TSYS,A$2000
LOCK FILER
```

** “addr” depends on the release date of **FILER**. Here are the values of “addr” for two different

* Unlike the patch to ProDOS, this patch need not be applied to the disk. You may wish simply to make the patch and execute the program. To do this, replace 3D0G with 2000G, and don’t BSAVE **FILER**. This patch works on the version of **FILER** released in 1984. It does not work with some pre-release versions, and may not work with future releases of **FILER**.

release dates:

Release date	address
1 JAN 84	4244
18 JUN 84	426A

FORCING PRODOS TO LOAD IN 48K

It is possible to load the ProDOS Kernel in main RAM (rather than in the bank switched memory or Language Card). In this case, you cannot use the BASIC Interpreter (since it is assembled for a fixed location which conflicts with the alternate location of the Kernel), or **EDASM.SYSTEM** from the toolkit package. You can, however, use other programs, such as the **EXERCISER** or **BUGBYTER**. Forcing a 48K load is sometimes useful even in a larger machine if you want to trace execution into the ProDOS Kernel itself using **BUGBYTER**. Under ordinary circumstances, as soon as the bank switched memory is enabled, the ROM monitor disappears and **BUGBYTER** goes berserk! If the Kernel is in main RAM, however, this does not occur. To force a 48K load you must first place a “**SYSTEM**” program (with type **SYS**) on the diskette to be booted (make a copy of **BUGBYTER** called **BUGBYTER.SYSTEM**). This must be the first file whose name ends with “**.SYSTEM**” in the Volume Directory. You can then apply the following patch and reboot.

```
BLOAD BUGBYTER
CREATE BUGBYTER.SYSTEM,TSYS
BSAVE BUGBYTER.SYSTEM,TSYS,A$2000,L7177
BLOAD PRODOS,TSYS,A$2000
CALL -151
23FC:A9 50
BSAVE PRODOS,TSYS,A$2000
```

Note that the value of **\$50** above is the **MACHID** desired (Apple II Plus with 48K). You may add to that the bits necessary for an 80-column card or Thunderclock if you like. You may wish to append your own program to the **BUGBYTER.SYSTEM** image before **BSAVE**ing it so that it will be available to you once the 48K system is loaded. You can do this by inserting a **BLOAD MYPGM,A\$3D00** after the **CREATE**, and changing the length of the **BSAVE** to accomodate **BUGBYTER** and your program combined. When you boot the 48K diskette, your program will be loaded at **\$3D00**, immediately following **BUGBYTER**.

Chapter 8

ProDOS Global Pages

Readers of *Beneath Apple DOS* may remember that Chapter 8 of that book was devoted to a detailed analysis of DOS program logic. The contents of that chapter comprised one quarter of the book, and represented a complete description of more than 10K of machine language code. Two factors have led to the approach taken in *Beneath Apple ProDOS*. First, ProDOS code is expected to be much more volatile than that of DOS. If material similar to Chapter 8 in *Beneath Apple DOS* had been published here, it would have quickly become obsolete because of reassemblies of the operating system components by Apple. Throughout its lifetime, DOS was only completely reassembled once—when the change was made from 3.1 to 3.2 in 1979. Our book documented a form of DOS in which most of the instructions had not “moved” in nearly five years! In contrast, before *Beneath Apple ProDOS* was published, two different versions of ProDOS were already being distributed—1.0.1 and 1.0.2. Although the differences between them are very minor, insertions of instructions and data have caused the shifting of major sections of code. Similar changes are expected in the future.

A second factor in the decision to shorten Chapter 8 involved the physical size of ProDOS. The equivalent components of ProDOS, compared to the DOS code covered in our earlier book, occupy over 22K of memory. A complete treatment of this code would be a book in and of itself. Coupled with the increased complexity of ProDOS which has resulted in longer chapters overall, as well as the previously mentioned volatility of the code, we decided that an in-depth coverage of ProDOS program logic did not belong here.

However, recognizing the importance of this material to many of our readers, a special supplement has been prepared that provides a detailed description of every piece of code and data within the major ProDOS components. It is available directly from Quality Software. Updated editions of this supplement will be available on a periodic basis as Apple releases new versions of ProDOS. In addition, any errata and changes to the main body of *Beneath Apple ProDOS* will be found in future supplements, eliminating the need to buy future editions of this book.

BASIC INTERPRETER GLOBAL PAGE

This page of memory is rigidly defined by the ProDOS BI. Fields given here will not move in later versions of ProDOS and may be referenced by external, user-written programs. Future additions to the global page may be made in areas which are marked “Not used”.

Address	Label	Contents
BE00-BE02	BIENTRY	JMP to WARMDOS (BI warmstart vector).
BE03-BE05	DOSCMD	JMP to SYNTAX (BI command line parse and execute).
BE06-BE08	EXTRNCMD	JMP to user-installed external command parser.
BE09-BE0B	ERROUT	JMP to BI error handler.
BE0C-BE0E	PRINTERR	JUMP to BI error message print routine. Place error number in A-register.
BE0F	ERRCODE	ProDOS error code (also at \$DE. Applesoft ONERR code).
BE10-BE1F	OUTVEC	Default output vector in monitor and for each slot (1-7).
BE20-BE2F	INVEC	Default input vector in monitor and for each slot (1-7).
BE30-BE31	VECTOUT	Current output vector.
BE32-BE33	VECTIN	Current input vector
BE34-BE35	VDOSIO	BI's output intercept address.
BE36-BE37		BI's input intercept address.
BE38-BE3B	VSYSIO	BI's internal redirection by STATE.
BE3C	DEFSLT	Default slot.
BE3D	DEFDRV	Default drive.
BE3E	PREGA	A-register save area.
BE3F	PREGX	X-register save area.
BE40	PREGY	Y-register save area.
BE41	DTRACE	Applesoft TRACE is enabled flag (MSB on).
BE42	STATE	Current intercept state. 0 = immediate command mode. > 0 = deferred.
BE43	EXACTV	EXEC file active flag (MSB on).
BE44	IFILACTV	READ file active flag (MSB on).
BE45	OFILACTV	WRITE file active flag (MSB on).
BE46	PFXACTV	PREFIX read active flag (MSB on).
BE47	DIRFLG	File being READ is a DIR file (MSB on).
BE48	EDIRFLG	End of directory flag (no longer used).
BE49	STRINGS	String space count used to determine when to garbage collect.
BE4A	TBUFPTR	Buffered WRITE data length.
BE4B	INPTR	Command line assembly length.

Address	Label	Contents
BE4C	CHRLAST	Previous output character (for recursion check).
BE4D	OPENCNT	Number of files open (not counting EXEC).
BE4E	EXFILE	EXEC file being closed flag (MSB on).
BE4F	CATFLAG	Line type to format next in DIR file READ.
BE50-BE51	XTRNADDR	External command handler address.
BE52	XLEN	Length of command name (less one).
BE53	XCNUM	Number of command: <div> <div>\$00 = external</div> <div>\$0A = OPEN</div> <div>\$14 = WRITE</div> <div>\$01 = IN#</div> <div>\$0B = READ</div> <div>\$15 = APPEND</div> <div>\$02 = PR#</div> <div>\$0C = SAVE</div> <div>\$16 = CREATE</div> <div>\$03 = CAT</div> <div>\$0D = BLOAD</div> <div>\$17 = DELETE</div> <div>\$04 = FRE</div> <div>\$0E = BSAVE</div> <div>\$18 = PREFIX</div> <div>\$05 = RUN</div> <div>\$0F = CHAIN</div> <div>\$19 = RENAME</div> <div>\$06 = BRUN</div> <div>\$10 = CLOSE</div> <div>\$1A = UNLOCK</div> <div>\$07 = EXEC</div> <div>\$11 = FLUSH</div> <div>\$1B = VERIFY</div> <div>\$08 = LOAD</div> <div>\$12 = NOMON</div> <div>\$1C = CATALOG</div> <div>\$09 = SAVE</div> <div>\$13 = STORE</div> <div>\$1D = RESTORE</div> <div>\$1E = POSITION</div> </div>
BE54-BE55	PBITS	Permitted command operand bits: <div> <div>\$8000 Prefix needed. Pathname optional.</div> <div>\$4000 Slot number only (PR# or IN#).</div> <div>\$2000 Deferred command.</div> <div>\$1000 File name optional.</div> <div>\$0800 If file does not exist, create it.</div> <div>\$0400 T: file type permitted.</div> <div>\$0200 Second file name required.</div> <div>\$0100 First file name required.</div> <div>\$0080 AD: address keyword permitted.</div> <div>\$0040 B: byte offset permitted</div> <div>\$0020 E: ending address permitted.</div> <div>\$0010 L: length permitted.</div> <div>\$0008 @: line number permitted.</div> <div>\$0004 S or D: slot/drive permitted.</div> <div>\$0002 F: field permitted.</div> <div>\$0001 R: record permitted.</div> <div>(V always permitted but ignored).</div> </div>
BE56-BE57	FBITS	Operands found on command line. Same bit assignments as above.

Address	Label	Contents
BE58-BE59	VADDR	A keyword value.
BE5A-BE5C	VBYTE	B keyword value.
BE5D-BE5E	VENDA	E keyword value.
BE5F-BE60	VLNTH	L keyword value.
BE61	V SLOT	S keyword value.
BE62	VDRIV	D keyword value.
BE63-BE64	VFELD	F keyword value.
BE65-BE66	VRECD	R keyword value.
BE67	VVOLM	V keyword value (ignored).
BE68-BE69	VLINE	@ keyword value.
BE6A	VTTYPE	T keyword value (in hex).
BE6B	VIO SLT	PR# or IN# slot number value.
BE6C-BE6D	VPATH1	Primary pathname buffer (address of length byte).
BE6E-BE6F	VPATH2	Secondary pathname buffer (address of length byte).
BE70-BE84	GOSYSTEM	Call the MLI using the parameter tables which follow.
BE85	SYSCALL	MLI call number for this call.
BE86-BE87	SYSPARM	Address of the MLI parameter list for this call.
BE88-BE8A		Return from MLI call.
BE8B-BE9E	BADCALL	MLI error return; translate error code to BI error number.
BE9F	BISPARE1	Not used.
BEA0-BEAB	SCREATE	CREATE parameter list.
BEAC-BEAE	SSGPRFX	GET_PREFIX, SET_PREFIX, DESTROY parameter list.
BEAF-BEB3	SRENAME	RENAME parameter list.
BEB4-BEC5	SSGINFO	GET_FILE_INFO, SET_FILE_INFO parameter list.
BEC6-BECA	SONLINE	ONLINE, SET_MARK, GET_MARK, SET_EOF, GET_EOF, SET_BUF, GET_BUF, QUIT parameter list.
BECB-BED0	SOPEN	OPEN parameter list.
BED1-BED4	SNEWLN	SET_NEWLINE parameter list.
BED5-BEDC	SREAD	READ, WRITE parameter list.
BEDD-BEDE	SCLOSE	CLOSE, FLUSH parameter list.
BEDF-BEF4	CCCSPARE	"COPYRIGHT APPLE, 1983"
BEF5-BEF7	GETBUFR	GETBUFR buffer allocation subroutine vector
BEF8-BEFA	FREEBUFR	FREEBUFR buffer free subroutine vector
BEFB		Original HIMEM MSB
BEFC-BEFF		Not used.

PRODOS SYSTEM GLOBAL PAGE — MLI GLOBAL PAGE

Portions of this page of memory are rigidly defined by the MLI and are unlikely to move in later versions of ProDOS. However, some portions are less stable and could change in future releases.

Address	Label	Contents
Jump Vectors		
BF00-BF02	ENTRY	JMP to MLI
BF03-BF05	JSPARE	Jump to system death code (via \$BFF6)
BF06-BF08	DATETIME	Jump to Date/Time routine (RTS if no clock).
BF09-BF0B	SYSERR	JMP to system error handler.
BF0C-BF0E	SYSDEATH	JMP to system death handler.
BF0F	SERR	System error number.
Device Information		
BF10-BF11	DEVADR01	Slot 0 reserved.
BF12-BF13	DEVADR11	Slot 1, drive 1 device driver address.
BF14-BF15	DEVADR21	Slot 2, drive 1 device driver address.
BF16-BF17	DEVADR31	Slot 3, drive 1 device driver address.
BF18-BF19	DEVADR41	Slot 4, drive 1 device driver address.
BF1A-BF1B	DEVADR51	Slot 5, drive 1 device driver address.
BF1C-BF1D	DEVADR61	Slot 6, drive 1 device driver address.
BF1E-BF1F	DEVADR71	Slot 7, drive 1 device driver address.
BF20-BF21	DEVADR02	Slot 0 reserved.
BF22-BF23	DEVADR12	Slot 1, drive 2 device driver address.
BF24-BF25	DEVADR22	Slot 2, drive 2 device driver address.
BF26-BF27	DEVADR32	/RAM device driver address (need extra 64K).
BF28-BF29	DEVADR42	Slot 4, drive 2 device driver address.
BF2A-BF2B	DEVADR52	Slot 5, drive 2 device driver address.
BF2C-BF2D	DEVADR62	Slot 6, drive 2 device driver address.
BF2E-BF2F	DEVADR72	Slot 7, drive 2 device driver address.
BF30	DEVNUM	Slot and drive (DSSS0000) of last device.
BF31	DEVCNT	Count (minus 1) of active devices.
BF32-BF3F	DEVLST	List of active devices (slot, drive and identification-DSSSIIII).
BF40-BF4F		Copyright notice.
BF50-BF55	IRQXITX	Switch in language card and call IRQ handler at \$FFD8.
BF56-BF57	TEMP	Temporary storage for IRQ code.
BF58-BF6F	BITMAP	Bitmap of low 48K of memory.

Address	Label	Contents
BF70-BF71	BUFFER1	Open file 1 buffer address.
BF72-BF73	BUFFER2	Open file 2 buffer address.
BF74-BF75	BUFFER3	Open file 3 buffer address.
BF76-BF77	BUFFER4	Open file 4 buffer address.
BF78-BF79	BUFFER5	Open file 5 buffer address.
BF7A-BF7B	BUFFER6	Open file 6 buffer address.
BF7C-BF7D	BUFFER7	Open file 7 buffer address.
BF7E-BF7F	BUFFER8	Open file 8 buffer address.
Interrupt Information		
BF80-BF81	INTRUPT1	Interrupt handler address (highest priority).
BF82-BF83	INTRUPT2	Interrupt handler address.
BF84-BF85	INTRUPT3	Interrupt handler address.
BF86-BF87	INTRUPT4	Interrupt handler address (lowest priority).
BF88	INTAREG	A-register save area.
BF89	INTXREG	X-register save area.
BF8A	INTYREG	Y-register save area.
BF8B	INTSREG	S-register save area.
BF8C	INTPREG	P-register save area.
BF8D	INTBANKID	Bank ID byte (ROM, RAM1, or RAM2).
BF8E-BF8F	INTADDR	Interrupt return address.
General System Information		
BF90-BF91	DATE	YYYYYYM MMMDDDD.
BF92-BF93	TIME	...HHHHH ..MMMMM.
BF94	LEVEL	Current file level.
BF95	BUBIT	Backup bit.
BF96-BF97	SPARE1	Currently unused.
BF98	MACHID	Machine ID Byte.
		00.. 0... II
		01.. 0... II+
		10.. 0... IIe
		11.. 0... III emulation
		00.. 1... Future expansion
		01.. 1... Future expansion
		10.. 1... IIc
		11.. 1... Future expansion
		..00 Unused
		..01 48K

Address	Label	Contents
		..10 64K
		..11 128K
	X.. Reserved
	0. No 80-column card
	1. 80-column card present
	0 No compatible clock
	1 Compatible clock present
BF99	SLTBYT	Slot ROM map (bit on indicates ROM present).
BF9A	PFIXPTR	Prefix flag (0 indicates no active prefix).
BF9B	MLIACTV	MLI active flag (1... indicates active).
BF9C-BF9D	CMDADR	Last MLI call return address.
BF9E	SAVEX	X-register save area for MLI calls.
BF9F	SAVEY	Y-register save area for MLI calls.
Language Card Bank Switching Routines		
BFA0-BFCF		Language card entry and exit routines.
	BFA0 EXIT	
	BFAA EXIT1	
	BFB5 EXIT2	
	BFB7 MLIENT1	
Interrupt Routines		
BFD0-BFF3		Interrupt entry and exit routines.
	BFD0 IRQXIT	
	BFDf IRQXIT1	
	BFE2 IRQXIT2	
	BFE7 ROMXIT	
	BFEB IRQENT	
Data		
BFF4	BNKBYT1	Storage for byte at \$E000.
BFF5	BNKBYT2	Storage for byte at \$D000.
BFF6-BFFB		Switch on language card and call system death handler (\$D1E4).
Version Information		
BFFC	IBAKVER	Minimum version of Kernel needed for this interpreter.
BFFD	IVERSION	Version number of this interpreter.
BFFE	KBAKVER	Minimum version of Kernel compatible with this Kernel.
BFFF	KVERSION	Version number of this Kernel.



The ProDOS garbage collector
starts his day

Appendix A

Example Programs

This section is intended to supply the reader with utility programs which can be used to examine and repair diskettes, as well as typical programming applications for ProDOS. These programs are provided in their source form to serve as examples of the programming necessary to interface practical programs to ProDOS. The reader who does not know assembly language may also benefit from these programs by entering them from the monitor in their binary form and saving them to disk for later use. The use of diskettes is assumed, although most of the programs will work with a hard disk or can be easily modified for this purpose. It is recommended that, until you are completely familiar with the operation of these programs, you should use them on an “expendable” diskette. None of the programs can physically damage a diskette, but they can, if used improperly, destroy the data on a diskette, requiring it to be reinitialized.

Seven programs are provided:

DUMP TRACK DUMP UTILITY

This is an example of how to access the disk drive directly through its I/O select addresses. **DUMP** may be used to dump to memory any given track in its raw, preformatted form. This can be useful both in understanding how disks are formatted, and in diagnosing clobbered diskettes. **DUMP** will only operate on a Disk II drive or its equivalent.

FORMAT REFORMAT A RANGE OF TRACKS

This program will initialize a single track or a range of tracks on a diskette. **FORMAT** is useful in restoring a track whose sectoring has been damaged without reinitializing the entire diskette. **FORMAT** will only operate on a Disk II drive or its equivalent.

ZAP DISK UPDATE UTILITY

This program is the backbone of any attempt to patch a disk directory back together. It is also useful in examining the structure of files stored on disk and in applying patches to files or ProDOS directly. **ZAP** allows its user to read, and optionally write, any block on a disk volume. As such, it serves as a good example of a program which issues direct block I/O calls to the MLI.

MAP MAP FREESPACE ON A VOLUME

MAP is written in BASIC and proves that direct block I/O can be done directly from a BASIC program as well as from assembly language. **MAP** reads the volume freespace bit map and displays a map of freespace versus blocks in use on the screen.

FIB FIND INDEX BLOCKS UTILITY

FIB may be used when a directory for a volume has been destroyed. It searches every block on a volume for what appear to be index blocks, printing the block number location of each index block it finds. Knowing the locations of the index blocks and employing **ZAP**, the user can patch together a new directory.

TYPE TYPE COMMAND

The **TYPE** command may be added to the ProDOS BI as a new command. It allows a user to type (display) the contents of a file to the screen or a printer. **TYPE** serves as an example of an external command handler.

DUMBTERM DUMB TERMINAL PROGRAM

DUMBTERM serves as an example of programming with interrupts. It implements a simple terminal emulator program, using a CCS 7710 serial interface card. Interrupts are used to fill a circular buffer, allowing higher baud rates to be used.

STORING THE PROGRAMS ON DISKETTE

The enterprising programmer may wish to key in the source code for each program into an assembler and assemble the programs onto disk. The Merlin Pro Assembler was used to produce the listings presented here, and interested programmers should consult the documentation for that assembler for more information on the pseudo-opcodes used. For the non-assembly language programmer, the binary object code of each program may be entered from the monitor using the following procedure.

The assembly language listings consist of columns of information as follows.

1. The address of some object code
2. The object code which should be stored there
3. The statement number
4. The statement itself

For example,

```
2000:A9 02    36 FIB    LDA #2 BLOCK = 2
```

indicates that the binary code “A902” should be stored at \$2000 and that this is statement 36. To enter a program in the monitor, the reader must type in each address and its corresponding object code. The following is an example of how to enter the FIB program.

```
CALL -151 (enter the monitor)  
2000:A9 02  
2002:8D E9 20  
2005:A9 00  
2007:8D EA 20  
...etc...  
20EB:00 00  
20ED:00 00  
BSAVE FIB,A$2000,L$EF
```

Note that if a line (such as line 2 in FIB) has no object bytes associated with it, it may be ignored. Also, never type in a four digit hex number, such as the ones found in FIB on lines 22 through 27 or the “2044” on line 41-type only two digit object code numbers.

When the program is to be run:

```
BLOAD FIB  
CALL -151  
2000G
```

The BSAVE commands which must be used with the other programs are:

```
BSAVE DUMP,A$2000,L$100  
BSAVE FORMAT,A$2000,L$51C  
BSAVE ZAP,A$2000,L$47  
BSAVE FIB,A$2000,L$EF  
BSAVE DUMBTERM,A$2000,L$F7  
BSAVE TYPE,A$2000,L$1B4
```

DUMP—TRACK DUMP UTILITY

The **DUMP** program will dump any track on a diskette in its raw, pre-nibbilized format, allowing the user to examine the sector address and data fields and the formatting of the track. This allows the inquisitive reader to examine his own diskettes to better understand the concepts presented in the preceeding chapters. **DUMP** may also be used to examine some protected disks to see how they differ from normal ones and to diagnose diskettes with clobbered sector address or data fields with the intention of recovering from disk I/O errors. The **DUMP** program serves as an example of direct use of the Disk II hardware from assembly language, with no use of ProDOS.

To use **DUMP**, first store the number of the track you wish dumped at location **\$2003**, the device number you wish to use at location **\$2004** (the program defaults to slot 6, drive 1), and begin execution at **\$2000**. **DUMP** will return to the monitor after displaying the first part of the track in hexadecimal on the screen. The entire track image is stored, starting at **\$4000**. For example:

```
BLOAD DUMP (Load the DUMP program)
CALL -151 (Get into the monitor from BASIC)
```

(Now insert the diskette to be **DUMP**ed)

```
2003:11 N 2000G (Store an 11 (track 17) in $2003, N
terminates the store command, go to
location $2000)
```

The output might look like this...

```
4000- D5 AA 96 AA AB AA BB AB (Start of sector address)
4008- AA AB BA DE AA E8 C0 FF
4510- 9E FF FF FF FF FF D5 AA (Start Of sector data)
4018- AD AE B2 9D AC AE 96 96 (Sector data)
...etc.
```

Quite often, a sector with an I/O error has only one bit which is in error, either in the address or data fields. A particularly patient programmer in some circumstances can determine the location of the error and devise a means to correct it.

```
1      *      DUMP -- TRACK DUMP UTILITY.
2      *      ORG      $2000

4      *
5      *
6      * DUMP:THIS PROGRAM WILL ALLOW USER TO DUMP AN ENTIRE
7      * TRACK IN ITS RAW FORM INTO MEMORY FOR EXAMINATION.
8      *
8      *
```

```

9      * INPUT: $2003 = TRACK TO BE READ (DEFAULTS TO $00)      *
10     *      $2004 = UNIT NUMBER (DEFAULTS TO $60)              *
11     *                                                         *
12     * OUTPUT:$4000 = ADDRESS OF TRACK IMAGE                    *
13     *                                                         *
14     * ENTRY POINT: $2000                                       *
15     *                                                         *
16     * PROGRAMMER: PIETER M LECHNER 5/29/84                     *
17     *                                                         *
18     *****

20     * ZPAGE DEFINITIONS

22     PTR      EQU    $0      WORK POINTER
23     A1L      EQU    $3C     MONITOR POINTER
24     A2L      EQU    $3E     MONITOR POINTER

26     * OTHER ADDRESSES

28     BUFFER   EQU    $4000    TRACK IMAGE AREA
29     DELAY    EQU    $FCA8    MONITOR DELAY ROUTINE
30     XAM      EQU    $FDB3    MONITOR HEX DUMP SUBRTN

32     * DISK I/O SELECTS

34     DRVSM0   EQU    $C080    STEP MOTOR POSITIONS
35     DRVSM1   EQU    $C081
36     DRVSM2   EQU    $C082
37     DRVSM4   EQU    $C084
38     DRVSM6   EQU    $C086
39     DRVOFF   EQU    $C088    TURN DRIVE OFF AFTER 6 REVS
40     DRVON    EQU    $C089    TURN DRIVE ON
41     DRVSL1   EQU    $C08A    SELECT DRIVE 1
42     DRVRD    EQU    $C08C    READ DATA LATCH
43     DRVRDM   EQU    $C08E    SET READ MODE

45     * RECALIBRATE AND POSITION THE ARM TO THE DESIRED TRACK

2000: 4C 0A 20 47     ENTRY    JMP    START      SKIP DATA

2003: 00          49     TRACK    DFB    $00      TRACK TO DUMP
2004: 60          50     UNITNUM  DFB    $60      UNIT NUMBER TO USE
2005: 60          51     SLOT     DFB    $60      SLOT NUMBER TO USE
2006: 00          52     DESTRK   DFB    $00      DESTINATION TRACK
2007: 00          53     CURTRK   DFB    $00      CURRENT TRACK
2008: 00          54     DELTA    DFB    $00      NUMBER OF TRACKS TO MOVE
2009: 00          55     FLAG     DFB    $00      DIRECTION & ODD/EVEN FLAGS

200A: AD 04 20 57     START    LDA    UNITNUM    GET UNIT NUMBER
200D: 48          58           PHA    SAVE      FOR LATER
200E: 29 70      59           AND    #$70      GET SLOT ONLY
2010: 8D 05 20 60           STA    SLOT
2013: AA          61           TAX    PUT       SLOT IN X REG
2014: 68          62           PLA
2015: 10 01      63           BPL    DRIVE1    SELECT DRIVE 1
2017: E8          64           INX    SELECT   DRIVE 2
2018: BD 8A C0 65     DRIVE1  LDA    DRVSL1,X  SELECT APPROPRIATE DRIVE
201B: AE 05 20 66           LDX    SLOT      GET SLOT
201E: BD 89 C0 67           LDA    DRVON,X    TURN DRIVE ON
2021: BD 8E C0 68           LDA    DRVRDM,X  INSURE READ MODE

2024: 20 82 20 70           JSR    RECALC     MOVE ARM TO TRACK 0

2027: AD 03 20 72           LDA    TRACK     GET TRACK TO READ
202A: 8D 06 20 73           STA    DESTRK

```

```

202D: 20 9F 20 74          JSR  ARMOVE      GO THERE

                76  *   PREPARE TO DUMP TRACK TO MEMORY

2030: A9 00 78          LDA  #<BUFFER  POINT AT DATA
2032: 85 00 79          STA  PTR
2034: A9 40 80          LDA  #>BUFFER
2036: 85 01 81          STA  PTR+1
2038: A0 00 82          LDY  #0

                84  *   START DUMPING AT THE BEGINNING OF A SECTOR ADDRESS
                85  *   FIELD OR A SECTOR DATA FIELD

203A: AE 05 20 87        LDX  SLOT
203D: BD 8C C0 88  LOOP1  LDA  DRVRD,X    WAIT FOR NEXT BYTE
2040: 10 FB 89          BPL  LOOP1
2042: C9 FF 90          CMP  #$FF        AUTOSYNC?
2044: D0 F7 91          BNE  LOOP1        NO, DON'T START IN MIDDLE
2046: BD 8C C0 92  LOOP2  LDA  DRVRD,X    WAIT FOR NEXT BYTE
2049: 10 FB 93          BPL  LOOP2
204B: C9 FF 94          CMP  #$FF        TWO AUTOSYNCS?
204D: D0 EE 95          BNE  LOOP1        NOT YET
204F: BD 8C C0 96  LOOP3  LDA  DRVRD,X
2052: 10 FB 97          BPL  LOOP3
2054: C9 FF 98          CMP  #$FF        STILL AUTOSYNCS?
2056: F0 F7 99          BEQ  LOOP3        YES, WAIT FOR DATA BYTE
2058: D0 05 100         BNE  LOOP4        ELSE, START STORING DATA

                102 *   ONCE ALIGNED, BEGIN COPYING THE TRACK TO MEMORY.
                103 *   COPY AT LEAST TWICE ITS LENGTH TO INSURE WE GET IT
                104 *   ALL.

205A: BD 8C C0 106  LOOPD  LDA  DRVRD,X    WAIT FOR NEXT DATA BYTE
205D: 10 FB 107        BPL  LOOPD
205F: 91 00 108  LOOP4  STA  (PTR),Y    STORE IN MEMORY
2061: E6 00 109        INC  PTR          BUMP POINTER
2063: D0 F5 110        BNE  LOOPD
2065: E6 01 111        INC  PTR+1        DONE $4000 BYTES YET?
2067: 10 F1 112        BPL  LOOPD        NO, CONTINUE
2069: AE 05 20 113        LDX  SLOT
206C: BD 88 C0 114        LDA  DRVOFF,X    TURN DRIVE OFF

                116 *   WHEN FINISHED, DUMP SOME OF TRACK IN HEX ON SCREEN

206F: A9 00 118          LDA  #<BUFFER  DUMP 4000.40AF
2071: 85 3C 119          STA  A1L
2073: A9 40 120          LDA  #>BUFFER
2075: 85 3D 121          STA  A1L+1
2077: A9 AF 122          LDA  #<BUFFER+$AF
2079: 85 3E 123          STA  A2L
207B: A9 40 124          LDA  #>BUFFER+$AF
207D: 85 3F 125          STA  A2L+1
207F: 4C B3 FD 126        JMP  XAM        EXIT VIA HEX DISPLAY

                128 *   RECALIBRATE ARM

2082: A9 30 130  RECALC  LDA  #$30        PRETENT TO BE ON TRACK 48
2084: 8D 07 20 131        STA  CURTRK
2087: A9 00 132          LDA  #$00        SELECT TRACK 00
2089: 8D 06 20 133        STA  DESTRK
208C: 20 9F 20 134        JSR  ARMOVE      GO THERE
208F: AE 05 20 135        LDX  SLOT        GET SLOT NUMBER
2092: BD 80 C0 136        LDA  DRVSM0,X    TURN ALL PHASES OFF
2095: BD 82 C0 137        LDA  DRVSM2,X
2098: BD 84 C0 138        LDA  DRVSM4,X

```

```

209B: BD 86 C0 139      LDA  DRVSM6,X
209E: 60              RTS   RETURN    TO CALLER

142  *  ARM MOVE ROUTINE

209F: A9 00          144  ARMOVE  LDA  #$00
20A1: 8D 09 20      145          STA  FLAG      INITIALIZE FLAG
20A4: AD 07 20      146          LDA  CURTRK    GET CURRENT TRACK
20A7: 38              147          SEC
20A8: ED 06 20      148          SBC  DESTRK    SUBTRACT DESTINATION TRACK
20AB: F0 36          149          BEQ  DONE     IF EQUAL THEN EXIT
20AD: B0 04          150          BCS  OK       POSITIVE RESULT? YES, GO ON
20AF: 49 FF          151          EOR  #$FF     MAKE RESULT POSITIVE
20B1: 69 01          152          ADC  #$01
20B3: 8D 08 20      153  OK      STA  DELTA     SAVE RESULT
20B6: 2E 09 20      154          ROL  FLAG     SET IN/OUT FLAG
20B9: 4E 07 20      155          LSR  CURTRK   ON ODD OR EVEN TRACK?
20BC: 2E 09 20      156          ROL  FLAG     PUT RESULT IN FLAG
20BF: 0E 09 20      157          ASL  FLAG     ADJUST FOR TABLE OFFSET
20C2: AC 09 20      158          LDY  FLAG     GET TABLE OFFSET
20C5: B9 F8 20      159  LOOP    LDA  TABLE,Y  GET PHASE TO TURN ON
20C8: 20 E4 20      160          JSR  PHASE
20CB: B9 F9 20      161          LDA  TABLE+1,Y GET NEXT PHASE TO TURN ON
20CE: 20 E4 20      162          JSR  PHASE
20D1: 98              163          TYA
20D2: 49 02          164          EOR  #$02     ADJUST OFFSET
20D4: A8              165          TAY
20D5: CE 08 20      166          DEC  DELTA     DECREMENT NUMBER OF TRACKS TO MOVE
20D8: AD 08 20      167          LDA  DELTA
20DB: D0 E8          168          BNE  LOOP     IF NOT DONE, DO ANOTHER
20DD: AD 06 20      169          LDA  DESTRK   UPDATE CURRENT TRACK WITH
20E0: 8D 07 20      170          STA  CURTRK   WHERE THE ARM IS NOW
20E3: 60              171  DONE   RTS   DONE,   RETURN TO CALLER

173  *  TURN A PHASE ON, WAIT THEN TURN IT OFF

20E4: 0D 05 20      175  PHASE  ORA  SLOT      ADD SLOT TO PHASE
20E7: AA              176          TAX
20E8: BD 81 C0      177          LDA  DRVSM1,X  TURN ON A PHASE
20EB: 20 F2 20      178          JSR  WAIT     WAIT FOR ARM TO SETTLE
20EE: BD 80 C0      179          LDA  DRVSM0,X  TURN OFF PHASE
20F1: 60              180          RTS   RETURN  TO CALLER

182  *  20 MILLISECOND DELAY ROUTINE

20F2: A9 56          184  WAIT   LDA  #$56      WAIT ABOUT 20 MILLISECONDS
20F4: 20 A8 FC      185          JSR  DELAY
20F7: 60              186          RTS   RETURN  TO CALLER

188  *  PHASE TABLE

20F8: 02 04 06      190  TABLE DFB  $02,$04,$06,$00
20FB: 00
20FC: 06 04 02      191          DFB  $06,$04,$02,$00
20FF: 00

```

--End assembly, 256 bytes, Errors: 0

FORMAT—REFORMAT A RANGE OF TRACKS

FORMAT can be used to selectively format a single track, a range of tracks or an entire diskette. While it is primarily meant to be educational, it can assist in repairing damaged diskettes. For example, if a single sector was damaged, it could be repaired by **FORMAT**ting the particular track on which it resides. To avoid losing data, all other sectors on the track should be read and copied to another diskette prior to re**FORMAT**ting. After **FORMAT** is run, they can be copied back to the repaired diskette and data can be written to the previously damaged sector.

Note that **FORMAT** has very limited error handling capabilities; in addition, it may not work well on drives that are out of adjustment (too fast or slow). The method used to do the formatting, that of building an image of the track in memory and then writing that image to the diskette, is similar to the method used by “nibble” copy programs.

To run **FORMAT**, store the starting track number at location \$2003, the ending track number at location \$2004, the volume number at location \$2005, and the device number at location \$2006, then begin execution at \$2000. **FORMAT** will return to the monitor upon completion. If a track cannot be formatted for some reason (e.g., physical damage, etc.), an error will be indicated. For example:

```
BLOAD FORMAT (Load the FORMAT program)
CALL -151 (Get into the monitor from BASIC)
```

(Now insert the diskette to be **FORMAT**ted)

```
2003:11 11 FE 60 N 2000G (Store an 11 (track 17) in $2003
store an 11 in $2004, store an FE
(volume 254) in $2005, Store a
60 (slot 6 drive 1) in $2006, N
terminates the store command, go
to location $2000)
```

The output might look like this:

FORMATTING TRACK 22

WARNING: **FORMAT** will destroy existing data on the diskette in the indicated drive without allowing the user an opportunity to abort the program. Be sure the diskette in the drive is the one you wish to **FORMAT**.


```

1          ORG    $2000

3          *****
4          *
5          * FORMAT: THIS PROGRAM WILL INITIALIZE A RANGE OF
6          *          TRACKS WITH ANY VOLUME NUMBER DESIRED.
7          *
8          * INPUT: $2003 = FIRST TRACK TO BE INITIALIZED
9          *          DEFAULT ($00)
10         *          $2004 = LAST TRACK TO BE INITIALIZED
11         *          DEFAULT ($22)
12         *          $2005 = VOLUME NUMBER
13         *          DEFAULT ($FE)
14         *          $2006 = UNIT NUMBER
15         *          DEFAULT ($60)
16         *
17         * ENTRY POINT: $2000
18         *
19         * PROGRAMMER: PIETER LECHNER 5/19/84
20         *
21         *****

23         * ZPAGE DEFINITIONS

25         PTR      EQU    $0          WORK POINTER
26         A1        EQU    $3C        MONITOR POINTERS
27         A2        EQU    $3E
28         A4        EQU    $42

30         * MONITOR ROUTINES

32         BS        EQU    $FC10      BACKSPACE
33         HOME      EQU    $FC58      CLEAR SCREEN
34         DELAY     EQU    $FCA8      DELAY ROUTINE
35         NXTA4     EQU    $FCB4      INCREMENT POINTERS (A4/A1)
36         NXTA1     EQU    $FCBA      INCREMENT POINTER (A1)
37         COUT      EQU    $FDED      CHARACTER OUTPUT
38         PRBYTE    EQU    $FDDB      HEX OUTPUT
39         MOVE      EQU    $FE2C      MOVE ROUTINE

41         * DISK I/O SELECTS

43         DRVSM0    EQU    $C080      STEP MOTOR POSITIONS
44         DRVSM1    EQU    $C081
45         DRVSM2    EQU    $C082
46         DRVSM4    EQU    $C084
47         DRVSM6    EQU    $C086
48         DRVOFF    EQU    $C088      TURN DRIVE OFF
49         DRVON     EQU    $C089      TURN DRIVE ON
50         DRVSL1    EQU    $C08A      SELECT DRIVE 1
51         DRVSL2    EQU    $C08B      SELECT DRIVE 2
52         DRVRD     EQU    $C08C      READ DATA LATCH
53         DRVWR     EQU    $C08D      WRITE DATA LATCH
54         DRVSRDM    EQU    $C08E      SET READ MODE
55         DRVWRM    EQU    $C08F      SET WRITE MODE

57         * FORMAT PROGRAM

2000: 4C 07 20 59  FORMAT    JMP    ENTRY    SKIP OVER DATA

2003: 00        61  STRK      DFB    $00      STARTING TRACK
2004: 22        62  ETRK      DFB    $22      ENDING TRACK
2005: FE        63  VOLNUM    DFB    $FE      VOLUME NUMBER
2006: 60        64  UNITNUM   DFB    $60      UNIT NUMBER

```

```

        66      *  RECALIBRATE ARM TO THE DESIRED TRACK

2007: 20 F7 22 68      ENTRY      JSR      SCREEN      CLEAR SCREEN/DISPLAY MESSAGE
200A: AD 06 20 69              LDA      UNITNUM      GET UNIT NUMBER
200D: 48              70              PHA      SAVE      FOR LATER
200E: 29 70              71              AND      #$70      GET SLOT ONLY
2010: 8D D6 24 72              STA      SLOT
2013: AA              73              TAX      PUT      SLOT IN X REG
2014: 68              74              PLA      CHECK     WHICH DRIVE TO USE
2015: 10 01              75              BPL      DRIVE1     SELECT DRIVE 1
2017: E8              76              INX      SELECT     DRIVE 2
2018: BD 8A C0 77      DRIVE1     LDA      DRVSL1,X     SELECT APPROPRIATE DRIVE
201B: AE D6 24 78              LDX      SLOT      GET SLOT
201E: BD 89 C0 79              LDA      DRVON,X      TURN DRIVE ON
2021: BD 8E C0 80              LDA      DRVDRM,X     INSURE READ MODE
2024: BD 8C C0 81              LDA      DRVDR,X

2027: 20 79 22 83              JSR      RECALC      RECALIBRATE ARM
202A: AD 03 20 84              LDA      STRK      GET STARTING TRACK
202D: 8D D9 24 85              STA      DESTRK
2030: 20 96 22 86              JSR      ARMOVE      GO THERE

        88      *  BUILD TRACK IMAGE IN MEMORY

2033: 20 3B 21 90              JSR      FILLDATA     FILL DATA AREA WITH $96'S
2036: AD 03 20 91              LDA      STRK
2039: 8D D7 24 92              STA      TRACK      INITIALIZE TRACK NUMBER
203C: A9 05 93              LDA      #$05
203E: 8D CE 24 94              STA      RETRYCNT     INITIALIZE RETRY COUNT
2041: A9 CB 95              LDA      #<END
2043: 8D D2 24 96              STA      CUREND
2046: A9 24 97              LDA      #>END      SAVE CURRENT LENGTH
2048: 8D D3 24 98              STA      CUREND+1     OF SECTOR IMAGE
204B: 38 99      LOOPA      SEC
204C: AD D2 24 100             LDA      CUREND      COMPUTE LENGTH OF
204F: E9 40 101             SBC      #<IMAGE     CURRENT SECTOR IMAGE
2051: 8D D0 24 102             STA      LENGTH
2054: AD D3 24 103             LDA      CUREND+1
2057: E9 23 104             SBC      #>IMAGE
2059: 8D D1 24 105             STA      LENGTH+1
205C: EE D0 24 106             INC      LENGTH
205F: 20 62 21 107             JSR      FINDSTART     COMPUTE START OF TRACK IMAGE
2062: 20 83 21 108             JSR      BLDGAP1      BUILD GAP1 (128 BYTES)
2065: 20 9F 21 109             JSR      BLDTRK      BUILD TRACK IMAGE IN MEMORY
2068: 20 C9 21 110      ANOTHER JSR      FIXTRK      UPDATE ADDRESS INFO

        112     *  INITIALIZE TRACK

206B: 20 06 23 114             JSR      PRTRK      PRINT TRACK NUMBER
206E: 20 B5 20 115             JSR      WRITE      WRITE A TRACK
2071: B0 1F 116             BCS      ERROR1     WRITE PROTECT ERROR
2073: 20 31 21 117             JSR      VERIFY     VERIFY THE TRACK
2076: 90 20 118             BCC      NEXT      IF OK DO ANOTHER
2078: 38 119             SEC      ELSE      ADJUST GAP SIZE
2079: AD D2 24 120             LDA      CUREND
207C: E9 02 121             SBC      #$02
207E: 8D D2 24 122             STA      CUREND
2081: AD D3 24 123             LDA      CUREND+1
2084: E9 00 124             SBC      #$00
2086: 8D D3 24 125             STA      CUREND+1
2089: CE CE 24 126             DEC      RETRYCNT     DECREMENT RETRY COUNT
208C: 10 BD 127             BPL      LOOPA      IF OK TRY AGAIN

        129     *  ERROR OCCURED

```

```

208E: A9 02      131          LDA    #$02
2090: D0 02      132          BNE    ERR2
2092: A9 01      133  ERROR1  LDA    #$01
2094: 20 13 23   134  ERR2    JSR    ERRHDL    PRINT ERROR MESSAGE
2097: 60         135          RTS     EXIT     PROGRAM

2098: EE D7 24   137  NEXT    INC     TRACK    INCREMENT TRACK NUMBER
209B: AD D7 24   138          LDA    TRACK
209E: CD 04 20   139          CMP    ETRK    COMPAIR WITH LAST TRACK TO DO
20A1: F0 02     140          BEQ    LASTONE  IF EQUAL DO LAST ONE
20A3: B0 09     141          BCS    FINISH   IF DONE THEN EXIT PROGRAM
20A5: 8D D9 24   142  LASTONE STA    DESTRK
20A8: 20 96 22   143          JSR    ARMOVE  GO TO DESIRED TRACK
20AB: 4C 68 20   144          JMP    ANOTHER GO BACK AND DO ANOTHER

146 * WHEN DONE, EXIT

20AE: AE D6 24   148  FINISH  LDX     SLOT    GET SLOT
20B1: BD 88 C0   149          LDA    DRVOFF,X  TURN DRIVE OFF
20B4: 60         150          RTS     EXIT     PROGRAM

152 * WRITE MEMORY TO DISK

20B5: A9 00      154  WRITE   LDA    #$00
20B7: 85 00      155          STA    PTR
20B9: AD D5 24   156          LDA    START+1  POINT AT START OF DATA
20BC: 85 01      157          STA    PTR+1
20BE: AC D4 24   158          LDY    START
20C1: AE D6 24   159          LDX    SLOT
20C4: 38         160          SEC     ASSUME    ERROR
20C5: BD 8D C0   161          LDA    DRVWR,X
20C8: BD 8E C0   162          LDA    DRVDRM,X  CHECK WRITE PROTECT STATUS
20CB: 30 30      163          BMI    WPERR
20CD: A9 FF      164          LDA    #$FF
20CF: 9D 8F C0   165          STA    DRVWRM,X  WRITE 1 $FF
20D2: DD 8C C0   166          CMP    DRVDR,X
20D5: EA         167          NOP
20D6: 4C E0 20   168          JMP    LOOP1
20D9: 49 80      169  ASYNC   EOR     #$80    TURN HIGH BIT ON
20DB: EA         170          NOP
20DC: EA         171          NOP
20DD: 4C E9 20   172          JMP    WRIT    DELAY EXTRA 8 CYCLES
20E0: 48         173  LOOP1   PHA
20E1: 68         174          PLA
20E2: B1 00      175  LOOP2   LDA    (PTR),Y  GET BYTE TO WRITE
20E4: C9 80      176          CMP    #$80    IS IT "SYNC" BYTE
20E6: 90 F1      177          BCC    ASYNC   YES, THEN MAKE ADJUSTMENT
20E8: EA         178          NOP
20E9: 9D 8D C0   179  WRIT    STA    DRVWR,X  WRITE A BYTE
20EC: DD 8C C0   180          CMP    DRVDR,X
20EF: C8         181          INY     INCREMENT  OFFSET
20F0: D0 EE      182          BNE    LOOP1
20F2: E6 01      183          INC    PTR+1    INCREMENT POINTER
20F4: 10 EC      184          BPL    LOOP2
20F6: BD 8E C0   185          LDA    DRVDRM,X
20F9: BD 8C C0   186          LDA    DRVDR,X  PUT IN READ MODE
20FC: 18         187          CLC
20FD: 60         188  WPERR   RTS

190 * PREPARE TO DUMP TRACK TO MEMORY

20FE: A9 00      192  DUMP    LDA    #$00    POINT AT DATA
2100: 85 00      193          STA    PTR
2102: A9 40      194          LDA    #$40
2104: 85 01      195          STA    PTR+1

```

```

2106: A0 00      196          LDY    #0

                198  *  START DUMPING AS SOON AS TWO SYNC BYTES FOUND

2108: AE D6 24  200          LDX    SLOT
210B: BD 8C C0  201  LOOP3    LDA    DRVRD,X    WAIT FOR NEXT BYTE
210E: 10 FB      202          BPL    LOOP3
2110: C9 FF      203          CMP    #$FF        AUTOSYNC?
2112: D0 F7      204          BNE    LOOP3        NO, DON'T START YET
2114: BD 8C C0  205  LOOP4    LDA    DRVRD,X    WAIT FOR NEXT BYTE
2117: 10 FB      206          BPL    LOOP4
2119: C9 FF      207          CMP    #$FF        TWO AUTOSYNCS?
211B: D0 EE      208          BNE    LOOP3        NOT YET

                210  *  ONCE ALIGNED, BEGIN COPYING THE TRACK TO MEMORY.
                211  *  COPY ENOUGH TO INSURE WE GET IT ALL

211D: BD 8C C0  213  LOOPD    LDA    DRVRD,X    WAIT FOR NEXT DATA BYTE
2120: 10 FB      214          BPL    LOOPD
2122: 91 00      215  LOOP5    STA    (PTR),Y    STORE IN MEMORY
2124: E6 00      216          INC    PTR        BUMP POINTER
2126: D0 F5      217          BNE    LOOPD
2128: E6 01      218          INC    PTR+1        DONE $2000 BYTES YET?
212A: A5 01      219          LDA    PTR+1
212C: C9 60      220          CMP    #$60        NO, CONTINUE
212E: D0 ED      221          BNE    LOOPD
2130: 60         222          RTS

                224  *  READ SECTOR ZERO TO VERIFY FORMATTING

2131: 20 FE 20  226  VERIFY    JSR    DUMP        DUMP TRACK TO MEMORY
2134: 20 F1 21  227          JSR    SETUP
2137: 20 39 22  228          JSR    COMPARE    CHECK WHAT WE JUST WROTE
213A: 60         229          RTS

                231  *  FILL DATA AREA WITH $96'S

213B: 18         233  FILLDATA CLC
213C: A9 57      234          LDA    #<DATA
213E: 85 3C      235          STA    A1        SET A1 TO DATA START
2140: 69 56      236          ADC    #<DATA LTH
2142: 85 3E      237          STA    A2        SET A1 TO DATA END
2144: A9 23      238          LDA    #>DATA
2146: 85 3D      239          STA    A1+1
2148: 69 01      240          ADC    #>DATA LTH
214A: 85 3F      241          STA    A2+1
214C: A9 96      242          LDA    #$96
214E: 8D CF 24  243          STA    BYTE        INDICATE FILL BYTE
2151: 20 55 21  244          JSR    FILL        CALL FILL ROUTINE
2154: 60         245          RTS

                247  *  FILL MEMORY WITH A CONSTANT

2155: A0 00      249  FILL     LDY    #$00        INITIALIZE OFFSET
2157: AD CF 24  250  LOOP     LDA    BYTE        GET BYTE TO USE
215A: 91 3C      251          STA    (A1),Y    STORE A BYTE
215C: 20 BA FC  252          JSR    NXTA1    CALL MONITOR INCREMENT
215F: 90 F6      253          BCC    LOOP    LOOP UNTIL DONE
2161: 60         254          RTS

                256  *  COMPUTE START OF TRACK IMAGE

2162: A0 03      258  FINDSTART LDY    #$03
2164: AD D0 24  259          LDA    LENGTH    MULTIPLY LENGTH BY 16
2167: 0A         260  MORE     ASL    A

```

```

2168: 2E D1 24 261      ROL    LENGTH+1
216B: 88          262      DEY
216C: 10 F9      263      BPL    MORE
216E: 8D D0 24 264      STA    LENGTH

2171: 38          266      SEC    SUBTRACT    IT FROM $7F80
2172: A9 80      267      LDA    #$80        TO FIND START
2174: ED D0 24 268      SBC    LENGTH
2177: 8D D4 24 269      STA    START
217A: A9 7F      270      LDA    #$7F
217C: ED D1 24 271      SBC    LENGTH+1
217F: 8D D5 24 272      STA    START+1
2182: 60          273      RTS

275 * BUILD GAP1 AT START OF TRACK IMAGE

2183: 18          277 BLDGAP1 CLC
2184: AD D4 24 278      LDA    START
2187: 85 3C      279      STA    A1          SET A1 TO START
2189: 69 80      280      ADC    #$80
218B: 85 3E      281      STA    A2          SET A2 TO START + $80
218D: AD D5 24 282      LDA    START+1
2190: 85 3D      283      STA    A1+1
2192: 69 00      284      ADC    #$00
2194: 85 3F      285      STA    A2+1
2196: A9 7F      286      LDA    #$7F        USE $7F FOR SYNC BYTE
2198: 8D CF 24 287      STA    BYTE
219B: 20 55 21 288      JSR    FILL        CALL FILL ROUTINE
219E: 60          289      RTS

291 * BUILD TRACK IMAGE USING SECTOR IMAGE

219F: AD D2 24 293 BLDTRK LDA    CUREND
21A2: 85 3E      294      STA    A2          SET A2 TO SECTOR IMAGE END
21A4: AD D3 24 295      LDA    CUREND+1
21A7: 85 3F      296      STA    A2+1
21A9: A5 3C      297      LDA    A1          SET A4 TO CURRENT POSITION
21AB: 85 42      298      STA    A4          IN TRACK IMAGE
21AD: A5 3D      299      LDA    A1+1
21AF: 85 43      300      STA    A4+1
21B1: A9 0F      301      LDA    #$0F        SET COUNT TO 16
21B3: 8D CC 24 302      STA    COUNT
21B6: A0 00      303      LDY    #$00        CLEAR Y FOR MOVE ROUTINE
21B8: A9 23      304 MORE2 LDA    #>IMAGE
21BA: 85 3C      305      STA    A1          SET A1 TO SECTOR IMAGE START
21BC: A9 23      306      LDA    #>IMAGE
21BE: 85 3D      307      STA    A1+1
21C0: 20 2C FE 308      JSR    MOVE        MOVE SECTOR IMAGE TO TRACK IMAGE
21C3: CE CC 24 309      DEC    COUNT        DECREMENT COUNT
21C6: 10 F0      310      BPL    MORE2        LOOP UNTIL WE HAVE 16 SECTORS
21C8: 60          311      RTS

313 * FIX ADDRESS INFORMATION IN TRACK IMAGE

21C9: A9 00      315 FIXTRK LDA    #$00
21CB: 8D D8 24 316      STA    SECTOR        START WITH SECTOR ZERO
21CE: AD D4 24 317      LDA    START
21D1: 85 3C      318      STA    A1          SET A1 TO TRACK IMAGE START
21D3: AD D5 24 319      LDA    START+1
21D6: 85 3D      320      STA    A1+1
21D8: A9 00      321      LDA    #$00
21DA: 85 3E      322      STA    A2          SET A2 TO $8000
21DC: A9 80      323      LDA    #$80
21DE: 85 3F      324      STA    A2+1
21E0: 20 22 22 325 AGAIN JSR    POSITION    POSITION TO ADDRESS INFO

```

```

21E3: 20 4C 22 326      JSR  CMPADD  UPDATE ADDRESS INFO
21E6: EE D8 24 327      INC  SECTOR  NEXT SECTOR
21E9: AD D8 24 328      LDA  SECTOR
21EC: C9 10 329      CMP  #$10    DONE ALL 16 YET?
21EE: D0 F0 330      BNE  AGAIN  IF NOT DO ANOTHER
21F0: 60 331      RTS

333  * SET POINTERS FOR VERIFY ROUTINE
334  * ON EXIT: A1 > 1ST ADDRESS FIELD IN TRACK IMAGE
335  *      A4 > 1ST ADDRESS FIELD IN LIVE DATA FROM DISK

21F1: A9 00 337  SETUP  LDA  #$00
21F3: 85 3C 338      STA  A1      SET A1 TO $4000
21F5: A9 40 339      LDA  #$40
21F7: 85 3D 340      STA  A1+1
21F9: A9 00 341      LDA  #$00
21FB: 85 3E 342      STA  A2      SET A2 TO $6000
21FD: A9 60 343      LDA  #$60
21FF: 85 3F 344      STA  A2+1
2201: 20 22 22 345      JSR  POSITION  LOCATE 1ST ADDRESS FIELD
2204: A5 3C 346      LDA  A1
2206: 85 42 347      STA  A4      SET A4 TO POSITION FOUND
2208: A5 3D 348      LDA  A1+1
220A: 85 43 349      STA  A4+1
220C: AD D4 24 350      LDA  START
220F: 85 3C 351      STA  A1      SET A1 TO START OF TRACK IMAGE
2211: AD D5 24 352      LDA  START+1
2214: 85 3D 353      STA  A1+1
2216: A9 FE 354      LDA  #$FE
2218: 85 3E 355      STA  A2      SET A2 TO $7FFE
221A: A9 7F 356      LDA  #$7F
221C: 85 3F 357      STA  A2+1
221E: 20 22 22 358      JSR  POSITION  LOCATE 1ST ADDRESS FIELD
2221: 60 359      RTS

361  * LOCATE AN ADDRESS FIELD

2222: A0 00 363  POSITION LDY  #$00  INITIALIZE OFFSET
2224: A2 02 364  POS2   LDX  #$02  INITIALIZE COUNT
2226: 20 BA FC 365  POS3   JSR  NXTA1  INCREMENT POINTER
2229: B0 0D 366      BCS  DONE  IF PAST DATA THEN EXIT
222B: B1 3C 367      LDA  (A1),Y  GET A BYTE
222D: DD DD 24 368      CMP  TABLE,X  CHECK IT IN TABLE
2230: D0 F2 369      BNE  POS2   NOT THERE, THEN TRY AGAIN
2232: CA 370      DEX  FOUND  ONE, COUNT IT
2233: 10 F1 371      BPL  POS3   GO UNTIL WE HAVE THREE
2235: 20 BA FC 372      JSR  NXTA1  POINT ONE PAST
2238: 60 373  DONE  RTS

375  * COMPAIR TWO AREAS OF MEMORY

2239: A0 00 377  COMPARE LDY  #$00  INITIALIZE OFFSET
223B: B1 3C 378  LOOPC  LDA  (A1),Y  GET A BYTE (TRACK IMAGE)
223D: 09 80 379      ORA  #$80  TURN 7F'S TO FF'S
223F: D1 42 380      CMP  (A4),Y  COMPAIR WITH DISK IMAGE
2241: D0 07 381      BNE  MISMATCH  EXIT ON MISMATCH
2243: 20 B4 FC 382      JSR  NXTA4  INCREMENT BOTH POINTERS
2246: 90 F3 383      BCC  LOOPC  LOOP UNTIL DONE
2248: 18 384      CLC  INDICATE  SUCCESS
2249: 60 385      RTS
224A: 38 386  MISMATCH SEC  INDICATE  ERROR
224B: 60 387      RTS

389  * COMPUTE ADDRESS FIELD INFORMATION AND STORE IN TRACK IMAGE

```

```

224C: AD 05 20 391  CMPADD  LDA  VOLNUM  GET VOLUME NUMBER
224F: 20 6B 22 392          JSR  COMPUTE  COMPUTE AND STORE IT
2252: AD D7 24 393          LDA  TRACK    GET CURRENT TRACK
2255: 20 6B 22 394          JSR  COMPUTE  COMPUTE AND STORE IT
2258: AD D8 24 395          LDA  SECTOR   GET CURRENT SECTOR
225B: 20 6B 22 396          JSR  COMPUTE
225E: AD 05 20 397          LDA  VOLNUM
2261: 4D D7 24 398          EOR  TRACK
2264: 4D D8 24 399          EOR  SECTOR   GET CHECKSUM
2267: 20 6B 22 400          JSR  COMPUTE
226A: 60          401          RTS

403  * NIBBLIZE A BYTE

226B: 48          405  COMPUTE  PHA  SAVE  A-REGISTER
226C: 4A          406          LSR  A      0ABCDEFGH H
226D: 09 AA      407          ORA  #$AA    1A1C1E1G
226F: 91 3C      408          STA  (A1),Y  STORE IT
2271: 68          409          PLA  ABCDEFGH
2272: 09 AA      410          ORA  #$AA    1B1D1F1H
2274: C8          411          INY
2275: 91 3C      412          STA  (A1),Y  STORE IT
2277: C8          413          INY
2278: 60          414          RTS

416  * RECALIBRATE DISK ARM

2279: A9 30      418  RECALC  LDA  #$30    PRETENT TO BE ON TRACK 48
227B: 8D DA 24  419          STA  CURTRK
227E: A9 00      420          LDA  #$00    SELECT TRACK 00
2280: 8D D9 24  421          STA  DESTRK
2283: 20 96 22  422          JSR  ARMOVE   GO THERE
2286: AE D6 24  423          LDX  SLOT     GET SLOT NUMBER
2289: BD 80 C0  424          LDA  DRVSM0,X  TURN ALL PHASES OFF
228C: BD 82 C0  425          LDA  DRVSM2,X
228F: BD 84 C0  426          LDA  DRVSM4,X
2292: BD 86 C0  427          LDA  DRVSM6,X
2295: 60          428          RTS  RETURN  TO CALLER

430  * ARM MOVE ROUTINE

2296: A9 00      432  ARMOVE  LDA  #$00
2298: 8D DC 24  433          STA  FLAG    INITIALIZE FLAG
229B: AD DA 24  434          LDA  CURTRK   GET CURRENT TRACK
229E: 38          435          SEC
229F: ED D9 24  436          SBC  DESTRK   SUBTRACT DESTINATION TRACK
22A2: F0 36      437          BEQ  DONE2  IF EQUAL THEN EXIT
22A4: B0 04      438          BCS  OK     POSITIVE RESULT? YES, GO ON
22A6: 49 FF      439          EOR  #$FF   MAKE RESULT POSITIVE
22A8: 69 01      440          ADC  #$01
22AA: 8D DB 24  441          STA  DELTA   SAVE RESULT
22AD: 2E DC 24  442          ROL  FLAG    SET IN/OUT FLAG
22B0: 4E DA 24  443          LSR  CURTRK  ON ODD OR EVEN TRACK?
22B3: 2E DC 24  444          ROL  FLAG    PUT RESULT IN FLAG
22B6: 0E DC 24  445          ASL  FLAG    ADJUST FOR TABLE OFFSET
22B9: AC DC 24  446          LDY  FLAG    GET TABLE OFFSET
22BC: B9 EF 22  447          LOOP6  LDA  PTABLE,Y  GET PHASE TO TURN ON
22BF: 20 DB 22  448          JSR  PHASE
22C2: B9 F0 22  449          LDA  PTABLE+1,Y  GET NEXT PHASE TO TURN ON
22C5: 20 DB 22  450          JSR  PHASE
22C8: 98          451          TYA
22C9: 49 02      452          EOR  #$02   ADJUST OFFSET
22CB: A8          453          TAY
22CC: CE DB 24  454          DEC  DELTA   DECREMENT NUMBER OF TRACKS TO MOVE
22CF: AD DB 24  455          LDA  DELTA

```

```

22D2: D0 E8      456          BNE    LOOP6      IF NOT DONE, DO ANOTHER
22D4: AD D9 24   457          LDA    DESTRK     UPDATE CURRENT TRACK WITH
22D7: 8D DA 24   458          STA    CURTRK     WHERE THE ARM IS NOW
22DA: 60         459  DONE2  RTS     DONE,      RETURN TO CALLER

461 * TURN A PHASE ON, WAIT THEN TURN IT OFF

22DB: 0D D6 24   463  PHASE  ORA     SLOT      ADD SLOT TO PHASE
22DE: AA         464          TAX
22DF: BD 81 C0   465          LDA    DRVSM1,X   TURN ON A PHASE
22E2: 20 E9 22   466          JSR    WAIT      WAIT FOR ARM TO SETTLE
22E5: BD 80 C0   467          LDA    DRVSM0,X   TURN OFF PHASE
22E8: 60         468          RTS    RETURN     TO CALLER

470 * 20 MILLISECOND DELAY ROUTINE

22E9: A9 56      472  WAIT    LDA    #$56      WAIT ABOUT 20 MILLISECONDS
22EB: 20 A8 FC   473          JSR    DELAY
22EE: 60         474          RTS    RETURN     TO CALLER

476 * PHASE TABLE

22EF: 02 04 06   478  PTABLE  DFB    $02,$04,$06,$00
22F2: 00
22F3: 06 04 02   479          DFB    $06,$04,$02,$00
22F6: 00

481 * CLEAR SCREEN AND DISPLAY MESSAGE

22F7: 20 58 FC   483  SCREEN  JSR    HOME      CLEAR SCREEN
22FA: A9 E0      484          LDA    #<MESSAGE
22FC: 85 00      485          STA    PTR      POINT AT MESSAGE
22FE: A9 24      486          LDA    #>MESSAGE
2300: 85 01      487          STA    PTR+1
2302: 20 33 23   488          JSR    PRINT     PRINT IT
2305: 60         489          RTS

491 * PRINT TRACK NUMBER

2306: AD D7 24   493  PRTRK   LDA    TRACK     GET TRACK NUMBER
2309: 20 DA FD   494          JSR    PRBYTE    PRINT IT
230C: 20 10 FC   495          JSR    BS
230F: 20 10 FC   496          JSR    BS      MOVE CURSOR BACK
2312: 60         497          RTS

499 * ERROR HANDLER

2313: C9 01      501  ERRHDL  CMP    #$01      IS IT ERROR #1
2315: D0 0A      502          BNE    SECOND    NO, THEN ASSUME #2
2317: A9 F3      503          LDA    #<MESSAGE1
2319: 85 00      504          STA    PTR      POINT AT MESSAGE 1
231B: A9 24      505          LDA    #>MESSAGE1
231D: 85 01      506          STA    PTR+1
231F: D0 08      507          BNE    PRINTIT   ALWAYS TAKEN
2321: A9 25      508  SECOND  LDA    #>MESSAGE2
2323: 85 00      509          STA    PTR      POINT AT MESSAGE 2
2325: A9 25      510          LDA    #>MESSAGE2
2327: 85 01      511          STA    PTR+1
2329: 20 33 23   512  PRINTIT JSR    PRINT     PRINT IT
232C: AE D6 24   513          LDX    SLOT      GET SLOT
232F: BD 88 C0   514          LDA    DRVOFF,X   TURN DRIVE OFF
2332: 60         515          RTS    EXIT      PROGRAM

517 * PRINT ROUTINE

```



```

2333: A0 00      519 PRINT LDY #000 INITIALIZE OFFSET
2335: B1 00      520 CHAR LDA (PTR),Y GET CHARACTER
2337: F0 06      521 BEQ TERMINATE IF ZERO THEN EXIT
2339: 20 ED FD   522 JSR COUT PRINT CHARACTER
233C: C8         523 INY
233D: D0 F6      524 BNE CHAR DO ANOTHER
233F: 60         525 TERMINATE RTS

                527 * DATA AREA

                529 IMAGE EQU *

2340: D5 AA 96   531 HEADER1 DFB $D5,$AA,$96

                533 ADDRESS EQU *
2343: AA AA      534 VOL DFB $AA,$AA
2345: AA AA      535 TRK DFB $AA,$AA
2347: AA AA      536 SEC DFB $AA,$AA
2349: AA AA      537 CHK DFB $AA,$AA

234B: DE AA EB   539 TRAILER1 DFB $DE,$AA,$EB

234E: 7F 7F 7F   541 GAP2 DFB $7F,$7F,$7F
2351: 7F 7F 7F   542 DFB $7F,$7F,$7F

2354: D5 AA AD   544 HEADER2 DFB $D5,$AA,$AD

2357: 00 00 00   546 DATA DS $56
235A: 00 00 00 00 00 00 00 00
2362: 00 00 00 00 00 00 00 00
236A: 00 00 00 00 00 00 00 00
2372: 00 00 00 00 00 00 00 00
237A: 00 00 00 00 00 00 00 00
2382: 00 00 00 00 00 00 00 00
238A: 00 00 00 00 00 00 00 00
2392: 00 00 00 00 00 00 00 00
239A: 00 00 00 00 00 00 00 00
23A2: 00 00 00 00 00 00 00 00
23AA: 00 00 00
23AD: 00 00 00   547 DS $100
23B0: 00 00 00 00 00 00 00 00
23B8: 00 00 00 00 00 00 00 00
23C0: 00 00 00 00 00 00 00 00
23C8: 00 00 00 00 00 00 00 00
23D0: 00 00 00 00 00 00 00 00
23D8: 00 00 00 00 00 00 00 00
23E0: 00 00 00 00 00 00 00 00
23E8: 00 00 00 00 00 00 00 00
23F0: 00 00 00 00 00 00 00 00
23F8: 00 00 00 00 00 00 00 00
2400: 00 00 00 00 00 00 00 00
2408: 00 00 00 00 00 00 00 00
2410: 00 00 00 00 00 00 00 00
2418: 00 00 00 00 00 00 00 00
2420: 00 00 00 00 00 00 00 00
2428: 00 00 00 00 00 00 00 00
2430: 00 00 00 00 00 00 00 00
2438: 00 00 00 00 00 00 00 00
2440: 00 00 00 00 00 00 00 00
2448: 00 00 00 00 00 00 00 00
2450: 00 00 00 00 00 00 00 00
2458: 00 00 00 00 00 00 00 00
2460: 00 00 00 00 00 00 00 00
2468: 00 00 00 00 00 00 00 00
2470: 00 00 00 00 00 00 00 00

```

```

2478: 00 00 00 00 00 00 00 00
2480: 00 00 00 00 00 00 00 00
2488: 00 00 00 00 00 00 00 00
2490: 00 00 00 00 00 00 00 00
2498: 00 00 00 00 00 00 00 00
24A0: 00 00 00 00 00 00 00 00
24A8: 00 00 00 00 00
24AD: 00          548      DS      $01
                549      DATAEND EQU    *-1
                550      DATALTH EQU    DATAEND-DATA

24AE: DE AA EB  552      TRAILER2 DFB    $DE,$AA,$EB

24B1: 7F 7F 7F  554      GAP3      DFB    $7F,$7F,$7F
24B4: 7F 7F 7F  555      DFB      $7F,$7F,$7F
24B7: 7F 7F 7F  556      DFB      $7F,$7F,$7F
24BA: 7F 7F 7F  557      DFB      $7F,$7F,$7F
24BD: 7F 7F 7F  558      DFB      $7F,$7F,$7F
24C0: 7F 7F 7F  559      DFB      $7F,$7F,$7F
24C3: 7F 7F 7F  560      DFB      $7F,$7F,$7F
24C6: 7F 7F 7F  561      DFB      $7F,$7F,$7F
24C9: 7F 7F 7F  562      DFB      $7F,$7F,$7F
                563      END      EQU    *-1
                564      LEN      EQU    END-IMAGE+1

24CC: 00 00      566      COUNT     DFB    $00,$00
24CE: 00          567      RETRYCNT  DFB    $00
24CF: 00          568      BYTE      DFB    $00
24D0: 00 00      569      LENGTH    DFB    $00,$00
24D2: 00 00      570      CUREND    DFB    $00,$00
24D4: 00 00      571      START      DFB    $00,$00
24D6: 60          572      SLOT       DFB    $60
24D7: 00          573      TRACK      DFB    $00
24D8: 00          574      SECTOR     DFB    $00
24D9: 00          575      DESTRK     DFB    $00      DESTINATION TRACK
24DA: 00          576      CURTRK     DFB    $00      CURRENT TRACK
24DB: 00          577      DELTA      DFB    $00      NUMBER OF TRACKS TO MOVE
24DC: 00          578      FLAG       DFB    $00      DIRECTION & ODD/EVEN FLAGS
24DD: 96 AA D5   579      TABLE     DFB    $96,$AA,$D5

24E0: C6 CF D2   581      MESSAGE    ASC    "FORMATTING TRACK  "
24E3: CD C1 D4 D4 C9 CE C7 A0
24EB: D4 D2 C1 C3 CB A0 A0
24F2: 00          582      DFB      $00
24F3: 8D          583      MESSAGE1   DFB    $8D
24F4: D7 D2 C9   584      ASC      "WRITE PROTECT ERROR"
24F7: D4 C5 A0 D0 D2 CF D4 C5
24FF: C3 D4 A0 C5 D2 D2 CF D2
2507: 87 00      585      DFB      $87,$00
2509: 8D          586      MESSAGE2   DFB    $8D
250A: D5 CE C1   587      ASC      "UNABLE TO FORMAT"
250D: C2 CC C5 A0 D4 CF A0 C6
2515: CF D2 CD C1 D4
251A: 87 00      588      DFB      $87,$00

--End assembly, 1308 bytes, Errors: 0

```

ZAP—DISK UPDATE UTILITY

The next step up the ladder from **DUMP** and **FORMAT** is accessing data on the diskette at the block level. The **ZAP** program allows its user to specify a block number to be read into memory. The user can then make changes to the image of the block in memory, and subsequently use **ZAP** to write the modified image back over the block on disk. **ZAP** is particularly useful when it is necessary to patch up a damaged directory. Its use in this regard will be covered in more detail when **FIB** is explained.

To use **ZAP**, store the number of the block you wish to access at **\$2007** and **\$2008**. Store the least significant byte of the number in **\$2007** and the most significant byte in **\$2008**. For example, the key block of the Volume Directory may be read by entering **2007:02 00**. **\$2009** should be initialized with either **\$80** to indicate that a sector is to be read into memory, or **\$81** to ask that memory be written out to the block on the disk. You may also specify the disk drive to be used (slot 6, drive 1 is assumed) by storing a hex value of **\$s0** at **\$2004**, where “s” is the slot to be used. If you wish to access drive 2 for a given slot, turn on the most significant bit in **\$2004** (e.g. slot 6, drive 2 would be **2004:E0**). An example to illustrate the use of **ZAP** follows.

CALL -151 (Get into the monitor)
BLOAD ZAP (Load the ZAP program)

(Now insert the diskette to be ZAPped)

2007:02 00 80 N 2000G (Store a 2 (key block of the Volume directory) in \$2007/8 and \$80 (read block) at \$2009. N ends the store command and 2000G runs ZAP.)

The output might look like this...

```
1000- 00 00 03 00 FA 55 53 45
1008- 52 53 2E 44 49 53 4B 00
1010- 00 00 00 00 00 00 00 00
1018- 00 00 00 00 00 00 00 00
...etc...
```

In the example above; if the byte at offset 6 (the second character of the volume name, “**USERS.DISK**”) is to be changed to “O”, the following would be entered.

1006:4F (Change +\$06 to \$4F (“O”))
2009:81 N 2000G (Change ZAP to write mode and do it)

Note that ZAP will remember the previous values in \$2004 through \$2009. If something is wrong with the block to be read or written (an I/O error, perhaps), ZAP will print an error message of the form:

RC = 2B

A return code of \$2B, in this case, means that the diskette was write protected and a write operation was attempted. Other error codes are \$27 (I/O error), and \$28 (no device connected). Refer to the documentation on READ_BLOCK and WRITE_BLOCK in Chapter 6 for more information on these errors.

```

1          ORG    $2000

3          *****
4          *
5          * ZAP:  THIS PROGRAM WILL ALLOW ITS USER TO READ/WRITE *
6          *          INDIVIDUAL BLOCKS FROM/TO THE DISKETTE      *
7          *
8          * INPUT: $2004  = UNIT NUMBER (DSS$0000)
9          *                  DEFAULTS TO SLOT 6, DRIVE 1 ($60)
10         *                  (SLOT 6, DRIVE 2 IS $E0)
11         *          $2005/6 = ADDRESS OF AREA IN MEMORY TO BE
12         *                  READ/WITTEN.
13         *                  DEFAULTS TO $1000
14         *          $2007/8 = BLOCK NUMBER TO BE READ/WITTEN
15         *                  DEFAULTS TO $0000
16         *          $2009  = OPERATION TO BE PERFORMED:
17         *                  $80 = READ BLOCK
18         *                  $81 = WRITE BLOCK
19         *                  DEFAULTS TO READ BLOCK.
20         *
21         * ENTRY POINT: $2000
22         *
23         * PROGRAMMER: DON D WORTH - 2/25/84
24         *
25         *****

27         *          FIXED LOCATIONS WE NEED

29         A1L      EQU    $3C          MONITOR POINTERS
30         A1H      EQU    $3D
31         A2L      EQU    $3E
32         A2H      EQU    $3F
33         MLI      EQU    $BF00        MACHINE LANGUAGE INTERFACE
34         COUT     EQU    $FDED        MONITOR PRINT VECTOR
35         PRBYTE   EQU    $FDDA        MONITOR PRINT HEX BYTE
36         XAM      EQU    $FDB3        MONITOR HEX DUMP SUBRTN

38         *          ENTRY POINT, JUMP AROUND PARMS

2000: 4C 0A 20 40 ZAP      JMP      START      BR AROUND DATA

42         *          MLI READ/WRITE BLOCK PARAMETER LIST

2003: 03         44 RWBLP   DFB     $03          PARM COUNT = 3
2004: 60         45         DFB     $60          UNIT NUMBER
2005: 00 10      46 BUFF    DW      $1000        BUFFER ADDRESS
2007: 00 00      47         DW      $0000        BLOCK NUMBER
2009: 80         48 OPER    DFB     $80          OPERATION TO BE PERFORMED

50         *          START OF CODE, CALL MLI

```

```

200A: AD 09 20 52  START      LDA  OPER      PASS OPERATION CODE
200D: 8D 13 20 53          STA  OP
2010: 20 00 BF 54          JSR  MLI      CALL MLI
2013: 00          55  OP      DFB  $00
2014: 03 20      56          DW   RWBLP
2016: 90 19      57          BCC  EXIT      ALL WENT WELL???

          59  *      IF ERROR OCCURS, PRINT MESSAGE

2018: 48          61          PHA          ; SAVE ERROR CODE
2019: A9 87      62          LDA  #$87      BEEP THE SPEAKER
201B: 20 ED FD 63          JSR  COUT
201E: A9 D2      64          LDA  #"R"      PRINT THE "RC="
2020: 20 ED FD 65          JSR  COUT
2023: A9 C3      66          LDA  #"C"
2025: 20 ED FD 67          JSR  COUT
2028: A9 BD      68          LDA  #"="
202A: 20 ED FD 69          JSR  COUT
202D: 68          70          PLA
202E: 4C DA FD 71          JMP   PRBYTE      PRINT THE HEX VALUE

          73  *      WHEN FINISHED, DUMP SOME OF BLOCK IN HEX

2031: 18          75  EXIT      CLC
2032: AD 05 20 76          LDA  BUFF      DUMP $2000-$20B7
2035: 85 3C      77          STA  A1L
2037: 69 AF      78          ADC  #$AF
2039: 85 3E      79          STA  A2L
203B: AD 06 20 80          LDA  BUFF+1
203E: 85 3D      81          STA  A1H
2040: 69 00      82          ADC  #0
2042: 85 3F      83          STA  A2H
2044: 4C B3 FD 84          JMP   XAM      EXIT VIA HEX DISPLAY

--End assembly, 71 bytes, Errors: 0

```

MAP—MAP FREESPACE ON A VOLUME

The **MAP** program is written in BASIC and calls a tiny assembly language subroutine to read blocks from a ProDOS volume. It first reads the Volume Directory key block to determine the length and location of the Volume Bit Map. It then reads the bit map and prints a map of the volume's freespace on the screen.

To run **MAP** against a disk volume, first **LOAD** the program into BASIC, place the disk to be **MAP**ped in slot 6, drive 1, and then **RUN** the program. The output from such a run might look like this:

[illegible]

The **MAP** program first reads a short machine language program from data statements and pokes it into memory at **\$300**. The machine language program is as follows.

```

0300:      PHA                      ; SAVE REGISTERS UPON ENTRY
0301:      TYA
0302:      PHA
0303:      TXA
0304:      PHA
0305:      JSR $BF00             CALL MLI
0308:      DFB $80               READ BLOCK CALL
0309:      DW $315              PARAMETERS AT $315
030B:      STA $314             SAVE RETURN CODE
030E:      PLA                  ; RESTORE REGISTERS
030F:      TAX
0310:      PLA
0311:      TAY
0312:      PLA
0313:      RTS                  ; AND RETURN TO BASIC PROGRAM
0314:      DFB $00              RETURN CODE SAVED HERE
0315:      DFB $03              3 PARAMETERS
0316:      DFB $60              SLOT 6, DRIVE 1
0317:      DW $4000             BLOCK BUFFER IS AT $4000
0319:      DW 0                 BLOCK NUMBER FILLED IN BY BASIC PGM

```

MAP then calls the subroutine (see lines 1000-1020) to read the Volume Directory key block (BN = 2). It obtains the length of the volume name from +4 (eliminating the **\$FØ** entry type), and peeks the volume name and prints it on the screen from +5 in the buffer. If the total number of blocks on the

volume (+41/42) is not 280, then the message "NOT A PRODOS DISKETTE" is printed. Otherwise, the first block of the Volume Bit Map is read. A loop is then entered (lines 365-440) Where each binary bit which is one in the bit map is counted and printed as a "." (free) and each that is zero is counted and printed as a "U" (in use). The totals for used and free blocks are then printed and the program exits. If an error occurs, it is printed in decimal and the program aborts execution. Possible errors are 39 (I/O error) and 40 (no device connected).

MAP will not currently work for volumes with more or less than 280 blocks but this can be easily changed by the reader.

```

10 REM
20 REM THIS PROGRAM PRINTS A MAP OF
30 REM A PRODOS DISKETTE VOLUME.
40 REM
50 REM PROGRAMMER: DON D WORTH 2/22/84
60 REM
70 DATA 72,152,72,138,72,32,0,191,128,21,3,141,20,3,104,170,104,168
75 DATA 104,96,0,3,96
80 REM
90 REM POKE BLOCK READ SUBROUTINE INTO MEMORY
95 REM
100 SB = 768: REM SB=ADDR OF SUBROUTINE
105 BF = 16384: REM BUFFER IS AT $4000
110 FOR I = SB TO SB + 22
120 READ X: POKE I,X
130 NEXT I
140 POKE I,0: POKE I + 1,BF / 256
150 BN = SB + 25:RC = SB + 20
160 REM
170 REM READ THE VOLUME DIRECTORY KEY BLOCK TO FIND THE BIT MAP
190 REM
200 POKE BN,2
210 GOSUB 1000
230 REM
240 REM PRINT THE VOLUME NAME
250 REM
260 L = PEEK (BF + 4) - 240
265 HOME : PRINT "FREESPACE MAP FOR VOLUME /";
270 FOR I = 1 TO L
280 PRINT CHR$ ( PEEK (BF + I + 4));
290 NEXT I
300 PRINT "/": PRINT
310 REM
320 REM LOCATE AND READ BIT MAP BLOCK
330 REM
340 IF PEEK (BF + 41) + PEEK (BF + 42) * 256 < > 280 THEN PRINT CHR$
(7);"NOT A PRODOS DISKETTE": END
350 POKE BN, PEEK (BF + 39): POKE BN + 1, PEEK (BF + 40)
360 GOSUB 1000
362 REM
363 REM PRINT BIT MAP
364 REM
365 U = 0:F = 0
370 FOR B = 0 TO 34
380 X = PEEK (B + BF)
390 FOR I = 1 TO 8
400 IF X > = 128 THEN X = X - 128: PRINT ".":F = F + 1: GOTO 420
410 PRINT "U":U = U + 1

```

```
420 X = X * 2
430 NEXT I
440 NEXT B
442 REM
443 REM FINISH UP
444 REM
450 PRINT : PRINT : PRINT "U=USED BLOCK      .=FREE BLOCK"
455 PRINT : PRINT "BLOCKS USED: ";U;"      BLOCKS FREE: ";F
460 END
1000 REM
1001 REM READ A BLOCK FROM DISK
1002 REM
1003 CALL SB
1010 IF PEEK (RC) = 0 THEN RETURN
1020 PRINT "I/O ERROR = "; PEEK (RC); CHR$ (7): END
```


FIB—FIND INDEX BLOCK UTILITY

From time to time one of your diskettes will develop an I/O error smack in the middle of a directory. When this occurs, any attempt to use the files described by that directory will result in an **I/O ERROR** message from ProDOS. Generally, when this happens, the data stored in the files on the diskette is still intact, only the pointers to the files are gone. If the data absolutely must be recovered, a knowledgeable Apple user can reconstruct the directory from scratch. Doing this involves finding the index blocks for each file, and then using **ZAP** to patch a directory entry into the Volume Directory for each file which is found. **FIB** is a utility which will scan a disk volume for index blocks. Although it may flag some blocks which are not index blocks as being such it will never miss a valid index block. Therefore, after running **FIB** the programmer must use **ZAP** to examine each block printed by **FIB** to see if it is really an index block. Additionally, **FIB** will find every index block image on the volume, even some which were for files which have since been deleted. Since it is difficult to determine which files are valid and which are old deleted files, it is usually necessary to restore all the files and copy them to another diskette, and later delete the duplicate or unwanted ones.

To run **FIB**, simply load the program and start execution at **\$20000**. **FIB** will print the block number of each block it finds which bears a resemblance to an index block. For example

CALL -151 (Get into the monitor)

BLOAD FIB (Load the FIB program)

(Now insert the disk to be scanned into Slot 6 Drive 1)

20000G (Run the FIB program on this diskette)

The output might look like this...

BLK=0008	BLK=0099
BLK=0027	BLK=00AF
BLK=0028	BLK=00B1
BLK=003C	BLK=00B4
BLK=006F	BLK=00B7
BLK=0097	

Here 11 possible files were found. Of course, if some of the lost files were seedlings, they will not be represented here (seedlings are very difficult to locate once their directory entry is gone). And if some files were tree files, then three or more of the above block numbers could refer to index blocks for a single file. Also, if only one of several directories for a volume is damaged,

some of the block numbers given may refer to files whose directory entries are still intact. If, after running **FIB**, you get an error message (**RC = xx**, see **ZAP** errors), you may need to reformat the offending track. Divide the block number by eight to determine which track has the error. An alternative is to use **ZAP** to copy all blocks without errors to another formatted disk and write zeroes on the blocks corresponding to I/O errors. In this way you can preserve undamaged blocks which are on the same track with damaged ones.

In the example above, **ZAP** should now be used to read block 8. At +\$00 and +\$100 are the LSB and MSB of the block number for the first data block of the file (assuming this is not the master index block for a tree file). This block can be read and examined to try to identify the file and its type. Usually a BASIC program can be identified (even though it is stored in tokenized form) from the text strings contained in the **PRINT** statements. An ASCII conversion chart (see page 16 in the *Apple II Reference Manual for IIe Only*) can be used to decode these character strings. Straight **TXT** type files will also contain ASCII text, with each line separated from the others with **\$0Ds** (carriage returns). **BIN** type files are the hardest to identify and recover since their original address and length attributes were lost along with the directory entry. If you cannot identify a file, assume it is **BAS** (Applesoft BASIC). If this assumption turns out to be incorrect, you can always go back and **ZAP** the file type in the directory to try something else. Given below is an example **ZAP** to the Volume Directory to create an entry for the file whose index block is **BLK = 0008**. This **ZAP** assumes that the Volume Directory itself was lost and that you are starting the entire volume from scratch. Do not perform this patch to a diskette which is only partially damaged as you will wipe out the remainder of the valid directory entries in the process.

CALL -151
BLOAD ZAP

(insert disk to be ZAPped)

1000:00 N 1001<1000.11FEM	(Zero entire block of memory)
1000:00 00 03 00 F5 46 49 58	(Store a dummy Volume
	Directory
1008:55 50	header for volume /FIXUP)
1020:00 00 C3 27 0D 00 00 06	
1028:00 18 01	
102B:24 46 49 4C 45	(Make sapling entry for
	"FILE")
103B:FC	(file is type BAS)
103C:08 00	(key block is 8)
1040:00 xx 00	(EOF mark, see below)

```

1049:E3                                (full access "unlocked")
104A:01 08                            (AUX_TYPE = $801 for BAS
                                     file)
1050:02 00                            (header pointer)
2007:02 00 81 N 2000G                (write new block image out as
                                     first Volume Directory block)

```

The “xx” above should be set to the number of non-zero block numbers found in the index block as a first cut at the end of file mark. If garbage is loaded at the end of the program, try a smaller number. You may be able to deduce the true EOF by examining the program image on disk. Remember that `AUX_TYPE` will be different for different file types. See Chapter 4 for more information.

As soon as the entry is created using the above procedure, the file should be immediately copied to another diskette. Do not attempt to use the file in place because the Volume Bit Map has not been updated and several other fields in the directory entry have been omitted. Also, you do not want to risk damaging other “lost” files on the disk. Repeat the above process for each index block found by FIB. As each file is recovered, it may be `RENAMED` to its original name on the new diskette. Once all the files have been copied to another disk, and successfully tested, the damaged disk may be re-initialized.

```

1          ORG    $2000

3          *****
4          *
5          * FIB:  THIS PROGRAM SCANS AN ENTIRE VOLUME, SEARCHING *
6          *      FOR WHAT APPEAR TO BE INDEX BLOCKS AND PRINTS *
7          *      THE BLOCK NUMBER OF EACH ONE IT FINDS. FIB WILL *
8          *      WORK FOR ANY SIZED VOLUME. SLOT 6, DRIVE 1 IS *
9          *      ASSUMED. *
10         * *
11         * INPUT: NONE *
12         * *
13         * ENTRY POINT: $2000 *
14         * *
15         * PROGRAMMER: DON D WORTH - 2/25/84 *
16         * *
17         *****

19         *      FIXED LOCATIONS WE NEED

21         PTR      EQU    $48          WORK POINTER
22         BUFFER   EQU    $1000        BLOCK BUFFER
23         MLI       EQU    $BF00        MACHINE LANGUAGE INTERFACE
24         COUT      EQU    $FDED        MONITOR PRINT VECTOR
25         PRBYTE    EQU    $FDDA        MONITOR PRINT HEX BYTE
26         XAM       EQU    $FDB3        MONITOR HEX DUMP SUBRTN

28         *      OFFSETS INTO VOL DIR HEADER

30         TOTBLK    EQU    $1029
31         BITMAP     EQU    $1027

33         *      ENTRY POINT, READ VOLUME DIRECTORY HEADER

2000: A9 02        35         FIB      LDA    #2          BLOCK = 2

```

2002:	8D E9 20	36		STA	BLOCK	
2005:	A9 00	37		LDA	#0	
2007:	8D EA 20	38		STA	BLOCK+1	
200A:	20 97 20	39		JSR	READ	READ VOL DIR BLOCK
200D:	B0 35	40		BCS	EXIT	IF ERROR, GIVE UP RIGHT NOW
200F:	AD 29 10	41		LDA	TOTBLK	
2012:	8D ED 20	42		STA	LAST	
2015:	AD 2A 10	43		LDA	TOTBLK+1	
2018:	8D EE 20	44		STA	LAST+1	
201B:	4A	45		LSR	A	COMPUTE TOTBLK/4096
201C:	4A	46		LSR	A	
201D:	4A	47		LSR	A	
201E:	4A	48		LSR	A	FOR NUM BIT MAP BLOCKS
201F:	38	49		SEC		; AND ADD ONE TO THAT
2020:	6D 27 10	50		ADC	BITMAP	
2023:	8D E9 20	51		STA	BLOCK	
2026:	8D EB 20	52		STA	FIRST	
2029:	A9 00	53		LDA	#0	
202B:	6D 28 10	54		ADC	BITMAP+1	
202E:	8D EA 20	55		STA	BLOCK+1	POINT PAST THE BITMAP
2031:	8D EC 20	56		STA	FIRST+1	
		58	*		SEE IF WE ARE AT END OF VOLUME	
2034:	AD EA 20	60	NEWBLK	LDA	BLOCK+1	WHEN WE REACH LAST BLOCK +1
2037:	CD EE 20	61		CMP	LAST+1	
203A:	D0 13	62		BNE	READIT	
203C:	AD E9 20	63		LDA	BLOCK	
203F:	CD ED 20	64		CMP	LAST	
2042:	D0 0B	65		BNE	READIT	
2044:	60	66	EXIT	RTS	EXIT	TO SYSTEM
2045:	EE E9 20	67	NXTBLK	INC	BLOCK	INCREMENT BLOCK COUNT
2048:	D0 EA	68		BNE	NEWBLK	
204A:	EE EA 20	69		INC	BLOCK+1	
204D:	D0 E5	70		BNE	NEWBLK	AND CONTINUE LOOP
		72	*		READ BLOCK AND CHECK FOR VALIDITY AS AN INDEX BLOCK	
204F:	20 97 20	74	READIT	JSR	READ	READ THIS BLOCK
2052:	B0 F1	75		BCS	NXTBLK	ERROR?
2054:	A0 00	77		LDY	#0	
2056:	B9 00 10	78	CHKNG1	LDA	BUFFER,Y	MAKE SURE ITS NOT ALL ZERO
2059:	19 00 11	79		ORA	BUFFER+\$100,Y	
205C:	D0 05	80		BNE	CHKNG2	
205E:	C8	81		INY		
205F:	D0 F5	82		BNE	CHKNG1	
2061:	F0 E2	83		BEQ	NXTBLK	IF SO, SKIP IT
2063:	A0 00	85	CHKNG2	LDY	#0	
2065:	B9 00 10	86	CHKING	LDA	BUFFER,Y	ALLOW ZERO ENTRIES
2068:	D0 05	87		BNE	COMPR	
206A:	B9 00 11	88		LDA	BUFFER+\$100,Y	
206D:	F0 0E	89		BEQ	BLKOK	
206F:	A2 00	90	COMPR	LDX	#0	CHECK AGAINST FIRST
2071:	20 86 20	91		JSR	CMP	DO 16 BIT COMPARE
2074:	90 CF	92		BCC	NXTBLK	TOO SMALL FOR BLOCK NUMBER
2076:	A2 02	93		LDX	#2	CHECK AGAINST LAST
2078:	20 86 20	94		JSR	CMP	
207B:	B0 C8	95		BCS	NXTBLK	TOO LARGE FOR BLOCK NUMBER
207D:	C8	96	BLKOK	INY		
207E:	D0 E5	97		BNE	CHKING	
2080:	20 BE 20	99		JSR	PBLOCK	FOUND A CANDIDATE, PRINT BLOCK NO.
2083:	4C 45 20	100		JMP	NXTBLK	THEN CONTINUE

```

102 *      CMP: 16 BIT COMPARE

2086: B9 00 11 104 CMP      LDA    BUFFER+$100,Y CHECK MSB
2089: DD EC 20 105          CMP    FIRST+1,X
208C: 90 08 106          BCC    RTS      ITS SMALLER
208E: D0 06 107          BNE    RTS      ITS BIGGER
2090: B9 00 10 108          LDA    BUFFER,Y CHECK LSB
2093: DD EB 20 109          CMP    FIRST,X
2096: 60 110 RTS          RTS

112 *      READ A BLOCK FROM DISK TO $1000

2097: 20 00 BF 114 READ     JSR    MLI      CALL MLI
209A: 80 115          DFB    $80      READ CALL
209B: E5 20 116          DW    RWBLP
209D: B0 01 117          BCS    ERROR    ERROR?
209F: 60 118 RTS          RTS

120 *      IF ERROR OCCURS, PRINT MESSAGE

20A0: 48 122 ERROR     PHA          ; SAVE ERROR CODE
20A1: A9 87 123          LDA    #$87    BEEP THE SPEAKER
20A3: 20 ED FD 124          JSR    COUT
20A6: A9 D2 125          LDA    #"R"    PRINT THE "RC="
20A8: 20 ED FD 126          JSR    COUT
20AB: A9 C3 127          LDA    #"C"
20AD: 20 ED FD 128          JSR    COUT
20B0: A9 BD 129          LDA    #"="
20B2: 20 ED FD 130          JSR    COUT
20B5: 68 131          PLA
20B6: 20 DA FD 132          JSR    PRBYTE  PRINT THE HEX VALUE
20B9: A9 A0 133          LDA    #$A0    PRINT A BLANK
20BB: 20 ED FD 134          JSR    COUT    AND FALL THRU TO PRINT BLOCK

136 *      PRINT CURRENT BLOCK NUMBER

20BE: A9 C2 138 PBLOCK   LDA    #"B"    PRINT "BLK="
20C0: 20 ED FD 139          JSR    COUT
20C3: A9 CC 140          LDA    #"L"
20C5: 20 ED FD 141          JSR    COUT
20C8: A9 CB 142          LDA    #"K"
20CA: 20 ED FD 143          JSR    COUT
20CD: A9 BD 144          LDA    #"="
20CF: 20 ED FD 145          JSR    COUT
20D2: AD EA 20 146          LDA    BLOCK+1 PRINT BLOCK NUMBER IN HEX
20D5: 20 DA FD 147          JSR    PRBYTE  PRINT MSB
20D8: AD E9 20 148          LDA    BLOCK
20DB: 20 DA FD 149          JSR    PRBYTE  AND LSB
20DE: A9 8D 150          LDA    #$8D
20E0: 20 ED FD 151          JSR    COUT    NEW LINE
20E3: 38 152          SEC
20E4: 60 153 RTS          RTS

155 *      MLI READ/WRITE BLOCK PARAMETER LIST

20E5: 03 157 RWBLP     DFB    $03      PARM COUNT = 3
20E6: 60 158 UNIT      DFB    $60      UNIT NUMBER
20E7: 00 10 159 BUFF    DW    BUFFER    BUFFER ADDRESS
20E9: 00 00 160 BLOCK   DW    $0000     BLOCK NUMBER

20EB: 00 00 162 FIRST   DW    $0000     FIRST BLOCK AFTER BIT MAP
20ED: 00 00 163 LAST    DW    $0000     LAST BLOCK ON DISK +1

```

--End assembly, 239 bytes, Errors: 0

TYPE—TYPE COMMAND

The **TYPE** program is an example of how to add commands to the ProDOS BASIC Interpreter. **TYPE** may be installed as a command by **BRUN**ning **TYPE** or using the “-” smart **RUN** command. Once installed, the user may enter:

TYPE filename[,Sslot][,Ddrive]

The BI will not recognize “**TYPE**” as one of its commands and will pass control to the installed external command handler. The handler will locate and open the file, read its contents and print them on the screen or output device. The user may suspend the listing with any keypress and resume it with any other. A **control-C** will abort the listing.

TYPE’s operation begins when it is **BRUN**. Its first task is to allocate a page of memory between the BI and its buffers. It will copy the resident part of its program into this page. **TYPE** next stores the address of the newly allocated page in the BI’s **EXTERNCMD** vector in the BI Global Page. Each time the BI sees a command it doesn’t recognize, it will call the address in the **EXTERNCMD** vector before treating it as an invalid command. The transient portion of **TYPE** finishes up by copying and relocating the fixed addresses in the resident portion up to its new home in the newly allocated BI buffer. The transient portion then exits to ProDOS.

When an unknown command line is encountered, control passes to the resident code at **TYPENT**. **TYPENT** compares the command to the string “**TYPE**”, and if there is a match, it claims the command and returns to the BI to allow it to parse the filename and any other keywords given. If no **SYNTAX ERROR** occurs, control returns from the BI at the label **TYPBAK**. (If **TYPENT** does not recognize the command, it passes control on to the original contents of **EXTERNCMD**, in case there are other external command handlers installed.) When control returns to **TYPBAK**, the MLI is called to open the file, using the BI’s General Purpose buffer at **HIMEM** for an I/O buffer. The file is then read, 256 bytes at a time using **\$200** for a data buffer, and its contents are copied to the **COUT** screen output vector. At End of File, **TYPENT** exits to the BI through the MLI **CLOSE** function call.

TYPE may be used as a model for small command handlers. It is written in such a way that it may coexist with numerous other external command handlers by preserving the original value it finds in the **EXTERNCMD** vector. Suggestions for additional external commands might include a file **COPY** command or a file hex/ASCII **DUMP** command. Note that if the installed, resident portion

is longer than 256 bytes, the relocation code will have to be rewritten and will be a bit more complex.

```

1  *****
2  *
3  * TYPE: WHEN BRUN, THIS PROGRAM INSTALLS AN EXTERNAL *
4  * PRODOS BASIC INTERPRETER COMMAND BETWEEN THE *
5  * BI AND ITS BUFFERS. THE NEW COMMAND IS *
6  * INVOKED AS FOLLOWS: *
7  *
8  * TYPE <PATHNAME>[,S#][,D#] *
9  *
10 * THE TYPE COMMAND COPYS THE CONTENTS OF THE *
11 * INDICATED FILE TO THE SCREEN. *
12 *
13 * THE RESIDENT PORTION OF THE TYPE COMMAND *
14 * REQUIRES ONLY 256 BYTES OF RAM. *
15 *
16 * PROGRAMMER: DON D WORTH - 2/21/84 *
17 *
18 *****

20 * FIXED LOCATIONS WE NEED

22 PTR EQU $48 WORK ZPAGE POINTER
23 HIMEM EQU $73 APPLESOFT HIMEM (START OF GP BUFFER)
24 IN EQU $200 INPUT LINE BUFFER
25 MLI EQU $BF00 MACHINE LANGUAGE INTERFACE
26 KBD EQU $C000 KEYBOARD LATCH
27 KBDSTB EQU $C010 KEYBOARD CLEAR STROBE
28 COUT EQU $FDED MONITOR PRINT VECTOR

30 * SELECTED THINGS FROM THE BI GLOBAL PAGE

32 ORG $BE00 START OF BI GLOBAL PAGE

BE00: 4C 00 00 34 BIENTRY JMP $0000 WARM ENTRY INTO PRODOS BI
BE03: 4C 00 00 35 DOSCMD JMP $0000 COMMAND EXECUTER
BE06: 4C 00 00 36 EXTCMD JMP $0000 EXTERNAL COMMAND VECTOR
BE09: 4C 00 00 37 ERROUT JMP $0000 EXIT WITH ERROR
BE0C: 4C 00 00 38 PRNTERR JMP $0000 PRINT ERROR MESSAGE
BE0F: 00 39 ERRCODE DFB 0 ERROR CODE

41 ORG $BE50
BE50: 00 00 42 XTADDR DW $0000 EXECUTION ADDRESS OF EXTERNAL CMD
BE52: 00 43 XLEN DFB 0 LENGTH OF CMD STRING -1
BE53: 00 44 XCNUM DFB 0 BI COMMAND NUMBER

46 FN1 EQU $01 FILE NAME EXPECTED
47 SD EQU $04 SLOT/DRIVE PERMITTED
BE54: 00 00 48 PBITS DW 0 ALLOWED PARAMETERS BITS
BE56: 00 00 49 FBITS DW 0 PARAMETERS FOUND BITS

51 ORG $BE6C
BE6C: 00 00 52 VPATH1 DW $0000 ADDR OF PATHNAME 1 BUFFER
BE6E: 00 00 53 VPATH2 DW $0000 ADDR OF PATHNAME 2 BUFFER

55 GOSYS EQU * MLI CALL HANDLER

57 ORG $BECB
BECB: 03 58 SOPEN DFB $03 OPEN PARAMETER LIST
BECC: 00 00 59 DW $0000
BECE: 00 00 60 OSYSBUF DW $0000 BUFFER ADDR
BED0: 00 61 OREFNUM DFB $00 REF NUM RETURNED

```

		63	ORG	\$BED5	
		64	SREAD	EQU	*
		65	SWRITE	EQU	*
BED5:	04	66	DFB	\$04	PARAM COUNT = 4
BED6:	00	67	RWRFNUM	DFB	\$00
BED7:	00 00	68	RWDATA	DW	\$0000
BED9:	00 00	69	RWCOUNT	DW	\$0000
BEDB:	00 00	70	RWTRANS	DW	\$0000
					TRUE LENGTH
		72	SCLOSE	EQU	*
		73	SFLUSH	EQU	*
BEDD:	01	74	DFB	\$01	PARAM COUNT = 1
BEDE:	00	75	CFRFNUM	DFB	\$00
					REFNUM
		77	ORG	\$BEF5	
BEF5:	4C 00 00	78	GETBUFR	JMP	\$0000
					ALLOCATE BI BUFFER
		80	ORG	\$2000	
		82	*	THIS PART OF THE PROGRAM GETS CONTROL WHEN THE BRUN COMMAND IS ISSUED. IT RELOCATES THE RESIDENT PART OF THE CODE TO THE TOP OF MEMORY	
		83	*		
		84	*		
2000:	A9 01	86	TYPE	LDA	#1
2002:	20 F5 BE	87		JSR	GETBUFR
2005:	90 10	88		BCC	GOTBUF
					WE NEED 1 PAGE BUY MEMORY FOR RESIDENT CODE GOT IT
2007:	A0 00	90		LDY	#0
2009:	B9 76 20	91	ERLP	LDA	MSG,Y
200C:	20 ED FD	92		JSR	COUT
200F:	C8	93		INY	
2010:	C9 8D	94		CMP	#\$8D
2012:	D0 F5	95		BNE	ERLP
2014:	4C 00 BE	96	EXIT	JMP	BIENTRY
					AND LEAVE
2017:	85 49	98	GOTBUF	STA	PTR+1
2019:	AE 08 BE	99		LDX	EXTCMD+2
201C:	8D 08 BE	100		STA	EXTCMD+2
201F:	8E 40 21	101		STX	NXTCOM+2
2022:	AE 07 BE	102		LDX	EXTCMD+1
2025:	8E 3F 21	103		STX	NXTCOM+1
2028:	A0 00	104		LDY	#0
202A:	84 48	105		STY	PTR
202C:	8C 07 BE	106		STY	EXTCMD+1
202F:	B9 00 21	108	COPY	LDA	TYPENT,Y
2032:	91 48	109		STA	(PTR),Y
2034:	8C 8F 20	110		STY	SAVE
2037:	48	111		PHA	
2038:	29 03	112		AND	#\$03
203A:	A8	113		TAY	
203B:	68	114		PLA	
203C:	4A	115		LSR	A
203D:	4A	116		LSR	A
203E:	AA	117		TAX	
203F:	BD 90 20	118		LDA	OPTAB,X
2042:	88	119	OPLOOP	DEY	
2043:	30 04	120		BMI	OPDONE
2045:	4A	121		LSR	A
2046:	4A	122		LSR	A
2047:	D0 F9	123		BNE	OPLOOP
2049:	29 03	124	OPDONE	AND	#\$03
204B:	F0 26	125		BEQ	COPIED
204D:	AA	126		TAX	
204E:	AC 8F 20	127		LDY	SAVE
					ISOLATE LENGTH ZERO LENGTH OP (\$FF) SIGNALS END RESTORE YREG


```

2051: E0 03      128      CPX      #3
2053: F0 0C      129      BEQ      RELOC      RELOCATE ALL 3 BYTE OPS
2055: C8         130      INY
2056: CA         131      DEX
2057: F0 D6      132      BEQ      COPY      1 BYTE OP?
2059: B9 00 21   133      LDA      TYPENT,Y    NO, THAT LEAVES 2 BYTE OPS
205C: 91 48      134      STA      (PTR),Y
205E: C8         135      INY
205F: D0 CE      136      BNE      COPY      CONTINUE COPYING

                138      *      RELOCATE ABSOLUTE ADDRESSES IN INSTRUCTIONS

2061: C8         140      RELOC      INY
2062: B9 00 21   141      LDA      TYPENT,Y    COPY LSB OF ADDR
2065: 91 48      142      STA      (PTR),Y
2067: C8         143      INY
2068: B9 00 21   144      LDA      TYPENT,Y
206B: C9 21      145      CMP      #>TYPENT    THIS ADDR WITHIN TYPENT?
206D: D0 ED      146      BNE      STMSB      NO
206F: A5 49      147      LDA      PTR+1      YES, USE MSB OF NEW HOME
2071: D0 E9      148      BNE      STMSB      ALWAYS TAKEN

                150      *      RESIDENT CODE HAS BEEN INSTALLED, WE CAN EXIT

2073: 4C 00 BE   152      COPIED      JMP      BIENTRY    DONE, EXIT

                154      *      INSTALLATION DATA

2076: CE CF A0   156      MSG      ASC      "NO ROOM FOR NEW COMMAND"
2079: D2 CF CF CD A0 C6 CF D2
2081: A0 CE C5 D7 A0 C3 CF CD
2089: CD C1 CE C4
208D: 87 8D      157      DFB      $87,$8D

208F: 00         159      SAVE      DFB      0      SAVE AREA FOR YREG

                161      *      EACH BYTE CONTAINS THE LENGTHS OF 4 6502 OPCODES
                162      *      FOR EXAMPLE, AT +0 IS A $59.
                163      *      IN BINARY $59 = 01011001 OR
                164      *      01 01 10 01 (1,1,2,1)
                165      *      THESE ARE THE LENGTHS (IN REVERSED ORDER) FOR BRK,
                166      *      ORA (N,X), AND TWO UNDEFINED OPCODES.

2090: 59 69 59   168      OPTAB      DFB      $59,$69,$59,$7D OPCODE LENGTH TABLE
2093: 7D
2094: 5A 69 5D   169      DFB      $5A,$69,$5D,$7D
2097: 7D
2098: 5B 6A 59   170      DFB      $5B,$6A,$59,$7F
209B: 7F
209C: 5A 69 5D   171      DFB      $5A,$69,$5D,$7D
209F: 7D
20A0: 59 69 59   172      DFB      $59,$69,$59,$7F
20A3: 7F
20A4: 5A 69 5D   173      DFB      $5A,$69,$5D,$7D
20A7: 7D
20A8: 59 69 59   174      DFB      $59,$69,$59,$7F
20AB: 7F
20AC: 5A 69 5D   175      DFB      $5A,$69,$5D,$7D
20AF: 7D
20B0: 59 6A 55   176      DFB      $59,$6A,$55,$7F
20B3: 7F
20B4: 5A 6A 5D   177      DFB      $5A,$6A,$5D,$5D
20B7: 5D
20B8: 6A 6A 59   178      DFB      $6A,$6A,$59,$7F
20BB: 7F

```

20BC: 5A 6A 5D	179	DFB	\$5A,\$6A,\$5D,\$7F	
20BF: 7F				
20C0: 5A 6A 59	180	DFB	\$5A,\$6A,\$59,\$7F	
20C3: 7F				
20C4: 5A 69 5D	181	DFB	\$5A,\$69,\$5D,\$7D	
20C7: 7D				
20C8: 5A 6A 59	182	DFB	\$5A,\$6A,\$59,\$7F	
20CB: 7F				
20CC: 5A 69 5D	183	DFB	\$5A,\$69,\$5D,\$3D ONLY \$FF GIVES 0 LEN	
20CF: 3D				
	185 *		NOW STARTS THE RESIDENT CODE WHICH IS MOVED TO HIGH	
	186 *		MEMORY.	
	188 *		TYPENT IS CALLED BY THE BI WHENEVER IT DOESN'T	
	189 *		RECOGNIZE A COMMAND. WE TAKE A LOOK AT IT SO SEE IF	
	190 *		IT MIGHT BE THE "TYPE" COMMAND.	
	192	ORG	TYPE+256	MUST BE PAGE ALIGNED
2100: D8	193	TYPENT	CLD	; IDENTIFY TO BI
2101: AD 6C BE	194	LDA	VPATH1	COPY COMMAND LINE PTR
2104: 85 48	195	STA	PTR	
2106: AD 6D BE	196	LDA	VPATH1+1	
2109: 85 49	197	STA	PTR+1	
210B: A0 01	198	LDY	#1	
210D: B1 48	199	COMPR	LDA (PTR),Y	COMPARE COMMAND STRING
210F: D9 AC 21	200	CMP	NAME-1,Y	TO "TYPE"
2112: D0 29	201	BNE	NOTIT	NOT IT, VECTOR ON
2114: C8	202	INY		
2115: C0 05	203	CPY	#5	
2117: 90 F4	204	BCC	COMPR	CHECK ALL 4 CHARS
	206 *		IT HAS BEEN DETERMINED THAT THIS COMMAND IS MINE.	
	207 *		RETURN TO THE BI TO PARSE THE PATHNAME OPERAND	
2119: 88	209	DEY		
211A: 8C 52 BE	210	STY	XLEN	STORE COMMAND LINE INDEX
211D: A9 00	211	LDA	#0	
211F: 8D 53 BE	212	STA	XCNUM	COMMAND NUMBER 0 (EXTERN)
2122: A9 04	213	LDA	#SD	SLOT/DRIVE PERMITTED
2124: 8D 55 BE	214	STA	PBITS+1	
2127: A9 01	215	LDA	#FN1	WE NEED PATHNAME1
2129: 8D 54 BE	216	STA	PBITS	
212C: AD 44 21	217	WHERE	LDA TYPBAK	<USED TO FIND TYPBAK>
212F: AD 2D 21	218	LDA	WHERE+1	TELL BI WHERE TO RETURN
2132: 8D 50 BE	219	STA	XTADDR	
2135: AD 2E 21	220	LDA	WHERE+2	
2138: 8D 51 BE	221	STA	XTADDR+1	
213B: 18	222	CLC		; INDICATE COMMAND WAS FOUND
213C: 60	223	RTS		; BACK TO BI FOR MORE PARSING
	225 *		IF WE DON'T CLAIM A COMMAND, PASS IT THROUGH TO ANY	
	226 *		OTHER EXTERNAL COMMAND HANDLERS	
213D: 38	228	NOTIT	SEC	; INDICATE COMMAND NOT FOUND
213E: 4C 00 00	229	NXTCOM	JMP \$0000	<OLD EXTCMD VECTOR>
2141: 4C 09 BE	231	TYPERR	JMP ERRROUT	VECTOR TO BI ERROR EXIT
	233 *		ONCE THE COMMAND HAS BEEN PARSED, THE BI CALLS THE	
	234 *		FOLLOWING CODE TO FINISH HANDLING THE COMMAND	
2144: A4 74	236	TYPBAK	LDY HIMEM+1	MSB OF BUFFER AREA
2146: 8C CF BE	237	STY	OSYSBUF+1	COPY TO OPEN LIST
2149: A9 00	238	LDA	#0	

```

214B: 8D CE BE 239      STA  OSYSBUF
214E: A9 C8 240      LDA  #C8
2150: 20 70 BE 241      JSR  GOSYS      MLI: OPEN
2153: B0 EC 242      BCS  TYPERR      OPEN THE FILE
2155: AD D0 BE 243      LDA  OREFNUM      ERROR?
2158: 8D D6 BE 244      STA  RWRFNUM      COPY REF NUM
215B: 8D DE BE 245      STA  CFRFNUM      TO READ LIST
                                   AND CLOSE LIST

247 *      FILE IS OPEN, READ 256 BYTES AT A TIME

215E: A0 00 249      TYPLP  LDY  #0
2160: 8C D7 BE 250      STY  RWDATA
2163: 8C D9 BE 251      STY  RWCOUNT
2166: C8 252      INY
2167: 8C DA BE 253      STY  RWCOUNT+1  256 BYTES AT A TIME
216A: C8 254      INY
216B: 8C D8 BE 255      STY  RWDATA+1  TO $200
216E: A9 CA 256      LDA  #CA      MLI: READ
2170: 20 70 BE 257      JSR  GOSYS      READ 256 BYTES TO $200
2173: 90 0E 258      BCC  TYPRT      ALL WENT WELL
2175: C9 05 259      CMP  #5      EOF ERROR?
2177: D0 C8 260      BNE  TYPERR      NO, REAL ERROR
2179: A9 8D 261      TYQUIT  LDA  #8D
217B: 20 ED FD 262      JSR  COUT      PRINT A FINAL NEWLINE
217E: A9 CC 263      LDA  #CC      MLI: CLOSE
2180: 4C 70 BE 264      JMP  GOSYS      EXIT THROUGH CLOSE
266 *      COPY READ BUFFER TO SCREEN

2183: A0 00 268      TYPRT  LDY  #0
2185: B9 00 02 269      TYPPLP LDA  $200,Y      COPY BYTE BY BYTE
2188: 09 80 270      ORA  #80      MSB ON FOR COUT
218A: 20 ED FD 271      JSR  COUT      TO OUTPUT VECTOR
218D: AD 00 C0 272      LDA  KBD      CHECK FOR INTERVENTION
2190: 10 13 273      BPL  TYPON      NOTHING, CONTINUE
2192: 8D 10 C0 274      STA  KBDSTB      CLEAR STROBE
2195: C9 83 275      CMP  #83      CONTROL-C?
2197: F0 E0 276      BEQ  TYQUIT      YES, EXIT NOW
2199: AD 00 C0 277      TYPWT  LDA  KBD      WAIT FOR A SECOND KEYPRESS
219C: 10 FB 278      BPL  TYPWT
219E: 8D 10 C0 279      STA  KBDSTB      CLEAR STROBE
21A1: C9 83 280      CMP  #83      CONTROL-C?
21A3: F0 D4 281      BEQ  TYQUIT      YES, ABORT
21A5: C8 282      TYPON  INY      ; ELSE, CONTINUE TYPING
21A6: CC DB BE 283      CPY  RWTRANS      UNTIL BUFFER EMPTY
21A9: D0 DA 284      BNE  TYPPLP
21AB: F0 B1 285      BEQ  TYPLP      THEN GO READ ANOTHER

21AD: 54 59 50 287      NAME  ASC  'TYPE'      COMMAND NAME
21B0: 45
21B1: FF FF FF 288      DFB  $FF,$FF,$FF END OF PROGRAM FLAGS

```

--End assembly, 435 bytes, Errors: 0

DUMBTerm—DUMB TERMINAL PROGRAM

DUMBTerm is an example of how to program under ProDOS using interrupts. **DUMBTerm** acts as a simple, line-at-a-time terminal emulation program which interfaces to a California Computer Systems CCS 7710 serial card. The same program can be written for an Apple Super Serial card (but interrupts are not as reliable for that card). The main portion of the program merely loops, checking the keyboard and the serial card for incoming data. If a keypress is found, it is sent out over the serial line, if incoming serial data is found, it is displayed on the screen.

The meat of the program lies within the communications subroutines in the last half of the listing. **COMINT** initializes the CCS card for interrupts after passing the address of its interrupt handler (**COMIRQ**) to ProDOS via the **ALLOC_INTERRUPT** MLI call. Each time an interrupt occurs, the **COMIRQ** handler is called by ProDOS and it examines the CCS status register to determine whether the interrupt was raised by the CCS card. If not, **COMIRQ** returns to ProDOS with the carry flag set to indicate that it is not claiming the interrupt. This gives other interrupt handlers a chance to service the interrupt. If the interrupt was generated by the CCS card and incoming data is available, a character is read and stored in a 256-byte circular buffer and **COMIRQ** exits to ProDOS.

The buffer is called circular because a pair of index pointers are used (start of data, end of data) to mark the actual data within the buffer and these pointers may wrap at the end of the buffer back to its beginning. Thus, conceptually the buffer has no beginning or end. This means that the main program may be doing something else but the interrupt routine can buffer up to 256 characters coming in from the serial port before it will lose data. If the main part of the program was ever vigilant and constantly checked for incoming serial data, there would be no need for an **interrupt exit**. However, each time the **COUT** screen output subroutine is called, there is a potential that control will not return before the next character is available. This is because the Apple scrolls the screen by moving every line up a byte at a time, one by one. The process of scrolling a 40-column screen lasts over one character time at 1200 baud (120 characters per second) on the serial port. Thus, without an interrupt exit, a character would be lost each time the screen is scrolled up one line.

Ideally this should be all there is to it. On an Apple II Plus, **DUMBTerm** works well under most circumstances and with most 80-column cards. Unfortunately this is not the case on an Apple IIe. Due to an error in programming the Apple IIe ROM, the entire process of scrolling the 40-column screen in **PR#0** mode

is disabled from interrupts! Thus the interrupt exit is useless in this mode. For 80-column scrolls, the ROM also disables interrupts while scrolling the bank switched text page, and the interrupt exit is again useless (at 1200 baud anyway). The only mode where the exit is reliable is the 40-column mode with **PR#3** (**control-Q**). There are ways of avoiding these problems for 1200 baud. One is to change the window size (so that the monitor has less data to scroll). This is done by storing a new bottom line value at **\$23**. In **PR#0** 40-column mode, this value should be **\$15**. In 80-column mode, it must be **\$0E**. Another solution would be to reproduce the scrolling code from the monitor into your own program and “sniff” for interrupts (i.e. enable for interrupts and disable again) more frequently than Apple does. It is also worth noting that some 80-column cards, such as the ALS Smarterm, “scroll” by moving a hardware “top of screen” pointer. No CPU time is required to scroll this way and terminal programs are much easier to write.

DUMBTERM is also an example of a simple Interpreter or System Program. It sets up the stack register and ProDOS version fields in the System Global Page upon entry, and it exits upon sensing a **control-C** keypress using the **MLI QUIT** call.

```

1          ORG    $2000

3          *****
4          *
5          * DUMBTERM: THIS PROGRAM ACTS AS A DUMB TERMINAL
6          *          THROUGH A CCS 7710 SERIAL CARD USING
7          *          THE PRODOS INTERRUPT HANDLER FOR INPUT
8          *          INTERRUPTS. THIS PROGRAM FOLLOWS THE RULES
9          *          FOR A PRODOS INTERPRETER.
10         *
11         * ASSUMPTIONS: CCS7710 CARD IN SLOT 1
12         *          8 DATA BITS, 1 STOP, NO PARITY
13         *          BAUD RATE SET BY DIP SWITCHES ON CARD
14         *
15         * ENTRY POINT: $2000
16         *
17         * PROGRAMMER: DON D WORTH 3/8/84
18         *
19         *****

21         BELL    EQU    $87          BELL CHARACTER
22         CR      EQU    $0D          RETURN
23         RETURN  EQU    $8D          PRINTABLE RETURN
24         SPACE   EQU    $A0          BLANK

26         *      ZPAGE DEFINITIONS

28         A1L     EQU    $3C          MONITOR POINTER
29         A1H     EQU    $3D
30         A2L     EQU    $3E          MONITOR POINTER
31         A2H     EQU    $3F

33         *      OTHER ADDRESSES

35         MLI     EQU    $BF00        PRODOS ENTRY POINT

```

```

36 MACHID EQU $BF98 PRODOS MACHINE ID
37 IBAKVER EQU $BFFC MLI VERSION WANTED
38 IVERS EQU $BFFD MY VERSION
39 KBD EQU $C000 KEYBOARD STROBE
40 KBDRES EQU $C010 KEYBOARD RESET
41 COUT EQU $FDED MONITOR OUTPUT SUBROUTINE
42 RDKEY EQU $FD0C MONITOR INPUT SUBROUTINE
43 HOME EQU $FC58 MONITOR HOME
44 TEXT EQU $FB39 SET TEXT MODE

46 * START OF DUMB TERMINAL EMULATOR

2000: A2 FF 48 DTERM LDX #$FF SET UP STACK POINTER
2002: 9A 49 TXS
2003: A9 00 50 LDA #0
2005: 8D FC BF 51 STA IBAKVER VERSION 0 FOR EVERYBODY
2008: 8D FD BF 52 STA IVERS
200B: 20 62 20 53 JSR COMINT INITIALIZE SERIAL PORT
200E: 20 58 FC 54 JSR HOME CLEAR SCREEN
2011: A9 02 55 LDA #$02
2013: 2C 98 BF 56 BIT MACHID 80 COLUMN CARD PRESENT?
2016: F0 03 57 BEQ BEGIN NO
2018: 20 00 C3 58 JSR $C300 INITIALIZE 80 COLUMN CARD

60 *
61 * NOTE: NO NEED TO SET BIT MAP HERE SINCE I AM NOT DOING
62 * DYNAMIC MEMORY ALLOCATION. ALSO, WE WILL LEAVE
63 * THE POWERUP BYTE SUCH THAT A RESET FORCES REBOOT.
64 *
65 BEGIN EQU *
66 *
67 * --- SPECIAL NOTE ---
68 *
69 * IF AN APPLE IIe IS USED, THERE IS A POSSIBILITY OF DATA
70 * ERRORS AT 1200 BAUD WHEN SCROLLING THE SCREEN. IF THIS
71 * OCCURS, INSERT THESE INSTRUCTIONS AT THIS POINT:
72 *
73 * LDA #$0E SET WINDOW TO 14 LINES HIGH
74 * STA $23
75 *
76 * (LDA #$15 IF NO 80 COLUMN CARD IN YOUR IIe.)
77 *

79 * MAIN TERMINAL LOOP

201B: AD F4 20 81 LOOP LDA ERRORS HAVE ANY ERRORS OCCURED?
201E: F0 0A 82 BEQ NOERRS NO
2020: A9 87 83 LDA #BELL YES, BEEP AT HIM
2022: 20 ED FD 84 JSR COUT
2025: A9 00 85 LDA #0 AND CLEAR ERROR COUNTER
2027: 8D F4 20 86 STA ERRORS
202A: AD 00 C0 87 NOERRS LDA KBD FIRST TEST KEYBOARD
202D: 10 0F 88 BPL NOKEY NOTHING YET?
202F: 29 7F 89 AND #$7F
2031: C9 03 90 CMP #3 CONTROL-C?
2033: F0 1D 91 BEQ EXIT
2035: AD 00 C0 92 LDA KBD RELOAD CHARACTER
2038: 8D 10 C0 93 STA KBDRES CLEAR KEYBOARD
203B: 20 CC 20 94 JSR COMOUT SEND THE CHARACTER OUT

203E: 20 B1 20 96 NOKEY JSR COMTIN TEST FOR AVAILABLE INPUT
2041: F0 D8 97 BEQ LOOP NOTHING, CHECK KEYBOARD AGAIN
2043: 20 BA 20 98 JSR COMINP GET NEXT INPUT CHARACTER
2046: 09 80 99 ORA #$80 MSB ON FOR OUTPUT
2048: C9 8A 100 CMP #$8A LINE FEED?

```

```

204A: F0 CF      101      BEQ    LOOP      YES, SKIP IT
204C: 20 ED FD   102      JSR    COUT     ELSE, PRINT IT
204F: 4C 1B 20   103      JMP     LOOP     AND CONTINUE LOOP

                105 *      IF CONTROL-C IS TYPED, EXIT

2052: 20 D9 20   107      EXIT    JSR    COMCLS    CLOSE DOWN COMM LINE
2055: 20 00 BF   108      JSR    MLI      MLI: QUIT CALL
2058: 65         109      DFB     $65
2059: 5B 20     110      DW      QPARMS
205B: 04        111      QPARMS DFB     4          QUIT PARMLIST
205C: 00        112      DFB     0
205D: 00 00     113      DW      0
205F: 00 00     114      DFB     0
2060: 00 00     115      DW      0

                117 *****
                118 *
                119 *      CCS 7710 COMMUNICATIONS SUBROUTINES *
                120 *
                121 *****

                123 CCCOM    EQU    $C09E      CCS COMMAND REG
                124 CCSTS    EQU    $C09E      CCS STATUS REG
                125 CCDTA    EQU    $C09F      CCS DATA REG

                127 *      COMINT: INITIALIZE THE COMM LINE

2062: A9 00     129      COMINT  LDA     #0
2064: 8D F5 20   130      STA     CIRC     START CIRCULAR BUFFER PTRS
2067: 8D F6 20   131      STA     CIRCE
206A: 20 00 BF   132      JSR    MLI      ALLOCATE INTERRUPT EXIT
206D: 40        133      DFB     $40
206E: F0 20     134      DW      APARMS
2070: A9 23     135      LDA     #$23      RESET ACIA
2072: 8D 9E C0   136      STA     CCCOM
2075: A9 15     137      LDA     #$15      8 BITS/1 STOP/NO PARITY/NO INTS
2077: 8D 9E C0   138      STA     CCCOM
207A: AE 9F C0   139      LDX     CCDTA    THROW AWAY ANY GARBAGE
207D: 09 80     140      ORA     #$80      ENABLE INTERRUPTS
207F: 8D 9E C0   141      STA     CCCOM
2082: 60        142      RTS

                144 *      COMIRQ: INTERRUPT EXIT

2083: AD 9E C0   146      COMIRQ  LDA     CCSTS    CHECK STATUS
2086: 30 02     147      BMI     COMME    INTERRUPT WAS FOR ME?
2088: 38        148      SEC
2089: 60        149      RTS
208A: 48        150      COMME    PHA
208B: 29 70     151      AND     #$70      ANY ERRORS OCCURED?
208D: F0 03     152      BEQ     COMNE    NO
208F: EE F4 20   153      INC     ERRORS   YES, BUMP ERROR COUNT
2092: 68        154      COMNE    PLA
2093: 29 01     155      AND     #$01      CHECK FOR INCOMING DATA
2095: F0 18     156      BEQ     CLAIM    NOTHING, IGNORE OTHER INTERRUPTS
2097: AD 9F C0   157      LDA     CCDTA    GET INCOMING BYTE
209A: AE F6 20   158      LDX     CIRCE
209D: 9D F7 20   159      STA     CIRC,X   STORE IT AT END OF BUFFER
20A0: E8        160      INX
20A1: 8E F6 20   161      STX     CIRCE    UPDATE END POINT
20A4: EC F5 20   162      CPX     CIRC     WRAPPED BACK TO START?
20A7: D0 06     163      BNE     CLAIM    NO, DID NOT OVERRUN
20A9: EE F4 20   164      INC     ERRORS   OVERRUN ERROR, BUMP COUNT
20AC: CE F6 20   165      DEC     CIRCE    BACK UP END POINT

```

```

20AF: 18      166 CLAIM   CLC           ; CLAIM THE INTERRUPT
20B0: 60      167 RTS

                169 *      COMTIN: TEST FOR AVAILABLE INPUT
                170 *      IF NEQ, DATA IS AVAILABLE

20B1: 78      172 COMTIN   SEI           ; DISABLE FROM INTERRUPTS
20B2: AE F5 20 173 LDX     CIRC$     CHECK CIRCULAR BUFFER
20B5: EC F6 20 174 CPX     CIRC$     SEE IF ITS EMPTY
20B8: 58      175 CLI           ; REENABLE
20B9: 60      176 RTS

                178 *      COMINP: WAIT FOR NEXT INPUT, RETURN IN AREG

20BA: 20 B1 20 180 COMINP   JSR     COMTIN   TEST STATUS
20BD: F0 FB    181 BEQ     COMINP   NOTHING YET?
20BF: 78      182 SEI           ; DISABLE TO MESS WITH CIRC
20C0: AE F5 20 183 LDX     CIRC$     GET INPUT CHARACTER
20C3: BD F7 20 184 LDA     CIRC,X
20C6: E8      185 INX
20C7: 8E F5 20 186 STX     CIRC$     BUMP START POINTER FORWARD
20CA: 58      187 CLI           ; REENABLE FOR INTERRUPTS
20CB: 60      188 RTS

                190 *      COMOUT: OUTPUT A BYTE FROM AREG

20CC: 48      192 COMOUT   PHA
20CD: AD 9E C0 193 COMOL   LDA     CCSTS     CHECK STATUS
20D0: 29 02    194 AND     #$02     ISOLATE TX BUFFER BIT
20D2: F0 F9    195 BEQ     COMOL     NOT READY YET
20D4: 68      196 COMOIT   PLA
20D5: 8D 9F C0 197 STA     CCDTA     SEND BYTE
20D8: 60      198 RTS

                200 *      COMCLS: CLOSE COMM PORT

20D9: A9 23    202 COMCLS   LDA     #$23     STOP INTERRUPTS/DTR OFF
20DB: 8D 9E C0 203 STA     CCCOM
20DE: 58      204 CLI           ; JUST FOR SAFETY SAKE
20DF: A9 01    205 LDA     #1
20E1: 8D F0 20 206 STA     APARMS     CHANGE PARMLIST
20E4: 20 00 BF 207 JSR     MLI
20E7: 41      208 DFB     $41     DEALLOC_INTERRUPT
20E8: F0 20    209 DW     APARMS
20EA: A9 02    210 LDA     #2
20EC: 8D F0 20 211 STA     APARMS     LEAVE THINGS AS I FOUND THEM
20EF: 60      212 RTS

                214 *      DATA

20F0: 02      216 APARMS   DFB     2      ALLOC_INTERRUPT PARMS
20F1: 00      217 DFB     0
20F2: 83 20    218 DW     COMIRQ

20F4: 00      220 ERRORS   DFB     0      ERROR STATISTICS
20F5: 00      221 CIRC$    DFB     0      START OF DATA IN CIRC
20F6: 00      222 CIRC$    DFB     0      END OF DATA IN CIRC
20F7: 00 00 00 223 CIRC     DS     256     CIRCULAR INPUT BUFFER
20FA: 00 00 00 00 00 00 00 00 00
2102: 00 00 00 00 00 00 00 00 00
210A: 00 00 00 00 00 00 00 00 00
2112: 00 00 00 00 00 00 00 00 00
211A: 00 00 00 00 00 00 00 00 00
2122: 00 00 00 00 00 00 00 00 00
212A: 00 00 00 00 00 00 00 00 00

```



```
2132: 00 00 00 00 00 00 00 00
213A: 00 00 00 00 00 00 00 00
2142: 00 00 00 00 00 00 00 00
214A: 00 00 00 00 00 00 00 00
2152: 00 00 00 00 00 00 00 00
215A: 00 00 00 00 00 00 00 00
2162: 00 00 00 00 00 00 00 00
216A: 00 00 00 00 00 00 00 00
2172: 00 00 00 00 00 00 00 00
217A: 00 00 00 00 00 00 00 00
2182: 00 00 00 00 00 00 00 00
218A: 00 00 00 00 00 00 00 00
2192: 00 00 00 00 00 00 00 00
219A: 00 00 00 00 00 00 00 00
21A2: 00 00 00 00 00 00 00 00
21AA: 00 00 00 00 00 00 00 00
21B2: 00 00 00 00 00 00 00 00
21BA: 00 00 00 00 00 00 00 00
21C2: 00 00 00 00 00 00 00 00
21CA: 00 00 00 00 00 00 00 00
21D2: 00 00 00 00 00 00 00 00
21DA: 00 00 00 00 00 00 00 00
21E2: 00 00 00 00 00 00 00 00
21EA: 00 00 00 00 00 00 00 00
21F2: 00 00 00 00 00
```

```
--End assembly, 503 bytes, Errors: 0
```



Protecting Against Software Pirates

Appendix B

Diskette Protection Schemes

Protected software, that software which is modified in some way to prevent it from being copied or duplicated, has existed since very early in the history of the Apple II. This was even true of tape based software before disk drives were widely used. It is not known who protected the first piece of Apple software, but it has become a widespread practice. So has the practice of copying or **breaking** protected software. It should be pointed out that the following discussion will not take sides in the sometimes controversial subject of software protection. Rather, it will provide an informative look at the methods used to protect software and how those methods have been circumvented. This seems appropriate since almost all protection schemes now involve a modified or custom disk operating system.

At this time, ProDOS is still relatively new and it is unclear if it will influence the current practice of protecting software. In that a ProDOS disk is identical to earlier operating systems (DOS 3.3) at a byte level, it is certainly possible and probable that protection will exist. However, since ProDOS can and will support other storage devices (i.e. hard disks etc.), and with the current trend in sharing data between different applications, additional challenges exist for software developers. It is possible that the percentage of protected software may decrease somewhat with the introduction of ProDOS. The following discussion will deal with software protection in general on the Apple II family of computers.

A BRIEF HISTORY OF APPLE SOFTWARE PROTECTION

The first protected software was tape based and appeared in the latter part of 1978, and protected disks followed shortly thereafter. Early protection schemes were often quite effective as there was relatively little technical information available. Almost any modification that rendered the normal means of copying useless was sufficient in most cases—most schemes did in fact consist of relatively minor changes to the normal format of data. Individuals were able to discover and disable these protection methods on a program by program

basis, with little or no thought given to some automated means of reproducing protected software.

It was not until perhaps a year later, in late 1979, that a significant event occurred in disk protection. An extremely popular product was introduced that employed a considerably improved protection method. This marked the beginning of an escalating battle between those protecting software and those trying to copy it. The protection methods used became more and more complex and involved, increasing time and expense for developers to create. The copiers were also increasing their efforts. Programs appeared that were designed to copy particular software products—a major development in that it defeated a great number of different schemes with a single basic technique. These programs are referred to as **nibble copiers** and were introduced in early 1981.

Throughout this process, it is clear that both sides made use of the work of their counterparts. Protection schemes started to reflect a working knowledge of breaking techniques, and were often designed to circumvent a particular method or copier. The people breaking protection methods were also studying the various methods employed to stop them and producing increasingly effective tools. This produced a kind of ebb and flow seen in many competitive areas where each side gains a temporary advantage only to see it lost. Nibble copiers have had numerous revisions to cope with advancements in protection methods.

Another significant milestone was the introduction of a **hardware card** that could copy software from the Apple's memory, thus bypassing most existing protection methods. While it is hard to single out advancements in protection methods, the mere presence of the numerous copy programs, hardware devices, bulletin boards, classes, and magazines aimed at defeating protection methods indicates the constant advancement of protection. Also, the fact that software developers continue to protect software in the face of escalating costs indicates protection is still cost effective.

The cycle will no doubt continue. As new sophisticated schemes are developed, they will be broken by equally sophisticated schemes.

PROTECTION METHODS

It seems reasonable at this time to say that it is impossible to protect a program on disk in such a way that it can't be broken. This is, in large part, due to the nature of the Apple computer and its disk drive. It is an extremely well documented machine, with numerous publications available on both hardware and software functions. It is indeed difficult to hide anything (necessary in

protecting software) from anyone who is willing to invest sufficient time to find it.

Most disk protection methods fall into two different types of schemes. The first involves **format alterations**, altering some portion of the disk from its normal format (Chapter 3 and Appendix C provide descriptions of the normal format). The second involves creating an identifiable mark or signature that can be used to verify the disk.

FORMAT ALTERATION

A great number of ways exist to alter the format of normal data. They range from a single byte change to an entirely different format. A special case is **changing the location** of data, and not necessarily the structure of the data itself. An early example of this was moving the directory information from its normal location to a different track altogether. Later, tracks themselves were moved when “half” tracks became popular (but data must be a full track apart from other data, a restriction imposed by hardware). Some disks now even use quarter tracks. Although these methods were effective for a while, most nibble copiers are equipped to handle them.

A more elaborate technique used is known as **spiral tracks**. Data is staggered on alternating half tracks producing, as its name indicates, a spiral of sorts. Each half track contains approximately one third of a track of data. If the relationship of the different segments is critical, this method of protection can be quite difficult to deal with. Several copy programs are capable of handling this, but may require parameters and additional time to reproduce a disk protected in this manner.

As with location changes, **format changes** range from simple to complex. Almost all early changes were merely minor modifications to existing operating systems. The most common change was a change to the code that would read and write the Address Field. This was reasonable because the Address Field is never rewritten, and the only special code required was the code to read the modified Address Field.

The Address Field normally starts with the bytes \$D5/\$AA/\$96. If any of these bytes were changed, a standard operating system would not be able to locate that particular Address Field, causing an error. After the Address Field comes the address information itself (volume, track, sector, and checksum). Some common techniques include changing the order of this information, doubling the sector numbers, or altering the checksum with some constant. Any of the above would cause an error on a standard operating system. The

Address Field ends with two closing bytes (\$DE/\$AA), which can be changed or switched also. Similar kinds of changes can be made to the Data Field. These techniques worked well until automated programs appeared.

The first automated programs were good but generally made the assumption that the data portions had been modified and that the various gaps between the data portions were normal. This prompted modification of the gaps and eventually a radically different format in an attempt to circumvent the copy programs. These formats generally involved either different numbers of otherwise normal sectors on a track, or special sectors with Address and Data Fields combined. As with other advancements, this worked well for a time, but current nibble copiers make as few assumptions about the data format as possible and can generally deal with such techniques.

Here is the standard DOS 3.3 RWTS code to find the first nibble of the Address Field:

```

B94F- BD 8C C0      LDA  $C08C,X
B952- 10 FB        BPL  $B94F
B954- C9 D5        CMP  #$D5      (2 cycles)
B956- D0 F0        BNE  $B948      (2)
B958- EA          NOP              (2)

```

Now imagine a slight variation of this code:

```

B94F- BD 8C C0      LDA  $C08C,X
B952- 10 FB        BPL  $B94F
B954- 4A          LSR              (2)
B955- C9 6A        CMP  #$6A      (2)
B957- D0 EF        BNE  $B948      (2)

```

This variation has three important properties: it fits in the same number of bytes as the original, it runs in the same number of cycles, and it matches the same \$D5 nibble. However, it will also match a \$D4 nibble, because the LSR + CMP effectively treats the low bit as a wildcard.

In binary: \$D5 = 1101 0101

After LSR: 0110 1010 = \$6A

In binary: \$D4 = 1101 0100

After LSR: 0110 1010 = \$6A

By changing these five bytes of code, we have a unified RWTS that can read protected disks (\$D4/\$AA/\$96) and unprotected disks (\$D5/\$AA/\$96).

Now consider a second, more complex variation:

B94F-	BD 8C C0	LDA	\$C08C,X	
B952-	10 FB	BPL	\$B94F	
B954-	C9 D5	CMP	#\$D5	(2 cycles)
B956-	D0 F0	BNE	\$B948	(2)
B958-	EA	NOP		(2)
B959-	BD 8C C0	LDA	\$C08C,X	(4)
B95C-	C9 D5	CMP	#\$D5	(2)
B95E-	F0 12	BEQ	\$B972	
B960-	BD 8C C0	LDA	\$C08C,X	
B963-	10 FB	BPL	\$B960	
B965-	C9 AA	CMP	#\$AA	
B967-	D0 DF	BNE	\$B948	
B969-	BD 8C C0	LDA	\$C08C,X	
B96C-	10 FB	BPL	\$B969	
B96E-	C9 96	CMP	#\$96	
B970-	D0 D6	BNE	\$B948	

This RWTS allows for two radically different ways of starting an Address Field. It starts by matching a \$D5 nibble, as usual. But instead of moving on to the second nibble immediately, at \$B959 it checks the data latch exactly once to see if it is still \$D5. In the absence of a new “1” bit, the data latch will hold its value long enough for the CMP at \$B95C to detect a “0” bit after the \$D5 nibble. If found, it branches to \$B972 to parse the rest of the address field; otherwise it continues with the rest of the standard prologue (\$AA/\$96). Once again, we have a unified RWTS that can read protected and unprotected disks.

A third variation targeted the epilogue at the end of the Address Field. This is normally a sequence of two nibbles, \$DE/\$AA. Here is the DOS 3.3 code to match them:

B98B-	BD 8C C0	LDA	\$C08C,X
B98E-	10 FB	BPL	\$B98B
B990-	C9 DE	CMP	#\$DE
B992-	D0 AE	BNE	\$B942
B994-	EA	NOP	
B995-	BD 8C C0	LDA	\$C08C,X
B998-	10 FB	BPL	\$B995
B99A-	C9 AA	CMP	#\$AA
B99C-	D0 A4	BNE	\$B942
B99E-	18	CLC	
B99F-	60	RTS	

But if we ignored the second \$AA nibble, it would free up enough space for some fun tricks (not guaranteed, actual fun may vary).

B98B-	BD 8C C0	LDA	\$C08C,X	
B98E-	10 FB	BPL	\$B98B	
B990-	C9 DE	CMP	#\$DE	
B992-	F0 0A	BEQ	\$B99E	
B994-	48	PHA		(3 cycles)

B995- 68	PLA	(4)
B996- BD 8C C0	LDA \$C08C,X	(4)
B999- C9 08	CMP #\$08	(2)
B99B- B0 A5	BCS \$B942	
B99D- EA	NOP	
B99E- 18	CLC	
B99F- 60	RTS	

It starts by looking for a \$DE nibble, as usual. If found, it immediately exits with carry bit clear, indicating success. But if the first epilogue nibble is not \$DE, we fall through to \$B994 and intentionally burn CPU cycles before checking the data latch exactly once at \$B999. At this point, enough time has elapsed for the data latch to read some, but not all, of the bits of the next nibble. In other words, its value is dependent on the number of “0” bits after the first nibble. Once again, we have a unified RWTS that can read protected and unprotected disks.

SIGNATURE

The earliest example of a signature was probably an unused track (track 3 was commonly “un”used). The software verifies the signature by trying to read a sector on the unused track. In an error occurred, the signature was verified. As simple as this seems now, it was reasonably effective. While this is a fairly obvious example of a signature, later methods were much more difficult to detect. In fact, most signatures have been uncovered by finding and examining the code that verified it. Once a method was known, an algorithm could be developed to deal with it.

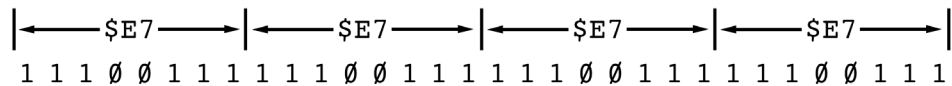
There are three common signatures used currently in protecting disks. The first to appear involves counting the number of bytes on a given track. This is commonly known as **nibble counting**. The reasoning was that no two drives could spin at precisely the same speed, and therefore would not reproduce a track precisely. While this is in fact true, a number of programs now provide the means to reproduce this type of signature.

Next to arrive was a method that was dependent on the positional relationship between different portions of the disk. This is commonly known as **synchronized tracks**. It generally involves reading a specific sector, then moving the disk arm to another track (often with nonstandard timing), and finding a particular sector first. The angle between the two sectors is arbitrary, but will always provide just enough time to move the arm and allow for any settling time needed. This relationship between tracks would not normally be maintained when copying the disk, and the signature would thus be removed.

This also is provided for in many current copying programs, sometimes requiring parameters for a particular disk.

The final method involves writing extra zero bits at given locations on disk. These can be thought of as **special sync bytes**. When the disk is read, these extra bits are normally discarded.

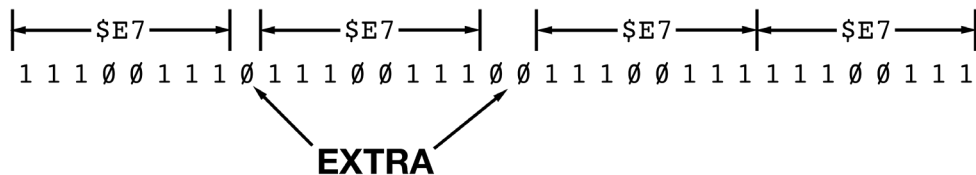
The most popular such bit pattern involved a sequence of **\$E7** nibbles. What would such a sequence look like on disk? The answer is: *It depends*. **\$E7** in hexadecimal is **11100111** in binary, so here is the simplest possible answer:



But wait. Every nibble read from disk must have its high bit set. In theory, we could insert one or two “0” bits after any of those nibbles. (Two is the maximum due to hardware limitations.) These extra “0” bits would be swallowed by the standard “wait for data latch to have its high bit set” loop, which we see over and over in any RWTS code:

```
:1    LDA $C08C,X
      BPL :1
```

Now consider the following bitstream:

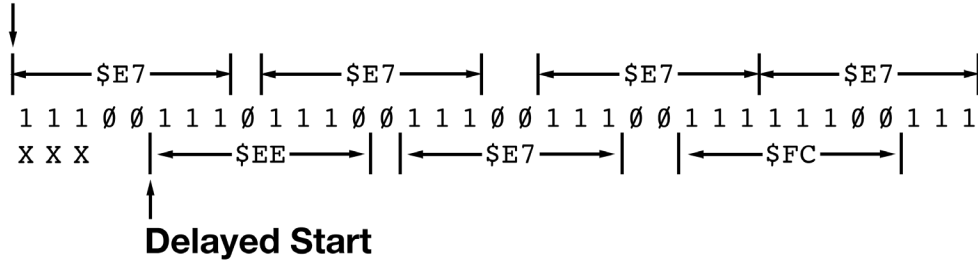


The first **\$E7** has one extra “0” bit after it, and the second **\$E7** has two extra “0” bits after it. Totally legal, works on any Apple II computer and any floppy drive. A **LDA \$C08C,X; BPL** loop would still interpret this bitstream as a sequence of four **\$E7** nibbles. Each of the extra “0” bits appear after we’ve just read a nibble and we’re waiting for the high bit to be set again. They get “swallowed”—ignored, like they were never there.

But what if we miss the first few bits of this bitstream, then start looking? The disk is always spinning, whether we’re reading from it or not. If we waste too much time doing something other than reading, we’ll literally miss some bits as the disk spins. (This is why the timing of low-level RWTS code is so critical.)

Let's say we waste 12 CPU cycles before we start reading this bitstream. Each bit takes 4 CPU cycles to go by, so after 12 cycles, we would have missed the first 3 bits (marked with an X).

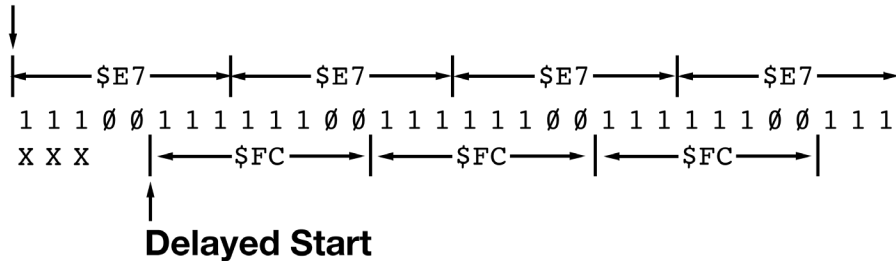
Normal Start



Now it's interpreted as a completely different nibble sequence, just by delaying a few CPU cycles before we start reading. Also note that some of those extra bits are no longer being ignored; now they're being interpreted as data, as part of the nibbles that are being returned to the higher level code. Meanwhile, other bits that used to be part of an \$E7 nibble are now being swallowed.

Now, let's go back to the first stream, which had no extra bits between the nibbles, and see what happens when we waste those same 12 CPU cycles.

Normal Start



After skipping the first three bits, the stream is interpreted as a series of \$FC \$FC \$FC repeating endlessly—not \$EE \$E7 \$FC like the other stream.

Nibble copiers eventually implemented routines to detect hidden “0” bits after any nibble, not just self-sync \$FF nibbles. But even the most sophisticated copier couldn't detect whether a nibble was followed by one or two “0” bits. By wasting just the right number of CPU cycles at just the right time, then interpreting the bits on the disk in mid-stream, developers could determine at runtime whether you had an original disk.

Here is the complete “\$E7 bitstream,” annotated to show both the synchronized and desynchronized nibble sequences.

$\leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$E7 \rightarrow \mid$
 1110011101110011100111001111100111001110011100111111001110011110011100111001110011111001111100111
 XXX $\leftarrow \$EE \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$FC \rightarrow \mid \leftarrow \$EE \rightarrow \mid \leftarrow \$E7 \rightarrow \mid \leftarrow \$FC \rightarrow \mid \leftarrow \$EE \rightarrow \mid \leftarrow \$EE \rightarrow \mid \leftarrow \$FC \rightarrow \mid$

We have dealt primarily with disk protection schemes and nibble copiers, but several other methods of protection exist. These are protection methods which do not allow a program to be taken out of memory and patched to disable the protection scheme. It is worth mentioning that copies produced by a nibble copier are themselves protected, but software broken in some other way may be copied by normal means.

MEMORY PROTECTION

It has long been realized that software is vulnerable as it is being loaded into memory, and when it resides entirely in memory. This has prompted a number of techniques, the earliest of which involved **reset protection**. When the Reset key was pressed (on early Apples), the software could be interrupted and was then resident in memory. Several memory locations were altered during a reset, and many programs were dependent on the values contained in those locations. The later Apple computers provide some measure of protection in that they make it much harder to interrupt software programs. The hardware boards designed to copy software from memory have made memory protection very difficult. The boards generate a Non-Maskable Interrupt and pass control to on-board software. It is not possible to prevent this interrupt from software. About the only defense is simply to never have the entire program in memory at one time. This is often inconvenient but may be the only effective defense.

CODE PROTECTION

Hiding the code that reads the unusual disk format or checks for a particular signature has become increasingly popular. Early schemes rarely tried to hide anything because there were few people who knew where to look or even what to look for. But it is clear now that most of the advancements in nibble copiers resulted from the examination of the actual code that provided the protection. Signature schemes would have been effective much longer if it had been possible to hide the code that verified them. While it is impossible to prevent the code from being found, it can be made more difficult. The general

method used is some sort of encryption of the code. It is decrypted just before execution, and either encrypted again or destroyed just after execution.

THE IDEAL PROTECTION SCHEME

There are thousands of programs available for the Apple II family of machines, and it is safe to say that they have all been copied despite a vast array of protection schemes. It seems reasonable to assume that this fact will not change. Nevertheless, it may be possible to devise a reasonably effective method. It would have to address the three primary ways that software is broken—nibble copiers, hardware boards that copy memory, and what we call the “front door” method.

NIBBLE COPIERS

Nibble copy programs have an advantage of sorts in that they need only respond to existing protection methods. This clearly requires considerable skill but not necessarily creativity. In fairness though it should be noted that at least one of the nibble copiers has included capabilities that may effectively deal with yet to be created protection schemes. The best that one should hope for is a protection method that requires parameters to be input by the user of the copier. If the method could be varied so that each variation required a different set of parameters, it would be considered a victory.

HARDWARE BOARDS

It is not possible through software to detect the presence of these boards, nor prevent them from saving an image of memory onto a disk. For this reason, they are particularly effective with programs that are totally loaded into memory and require no additional disk accesses. The only good defense is to never have the entire program in memory at one time. While this could create some difficulties such as decreased performance for particular programs, it is nevertheless necessary for single program products. Modular software requiring constant disk access may already provide sufficient protection.

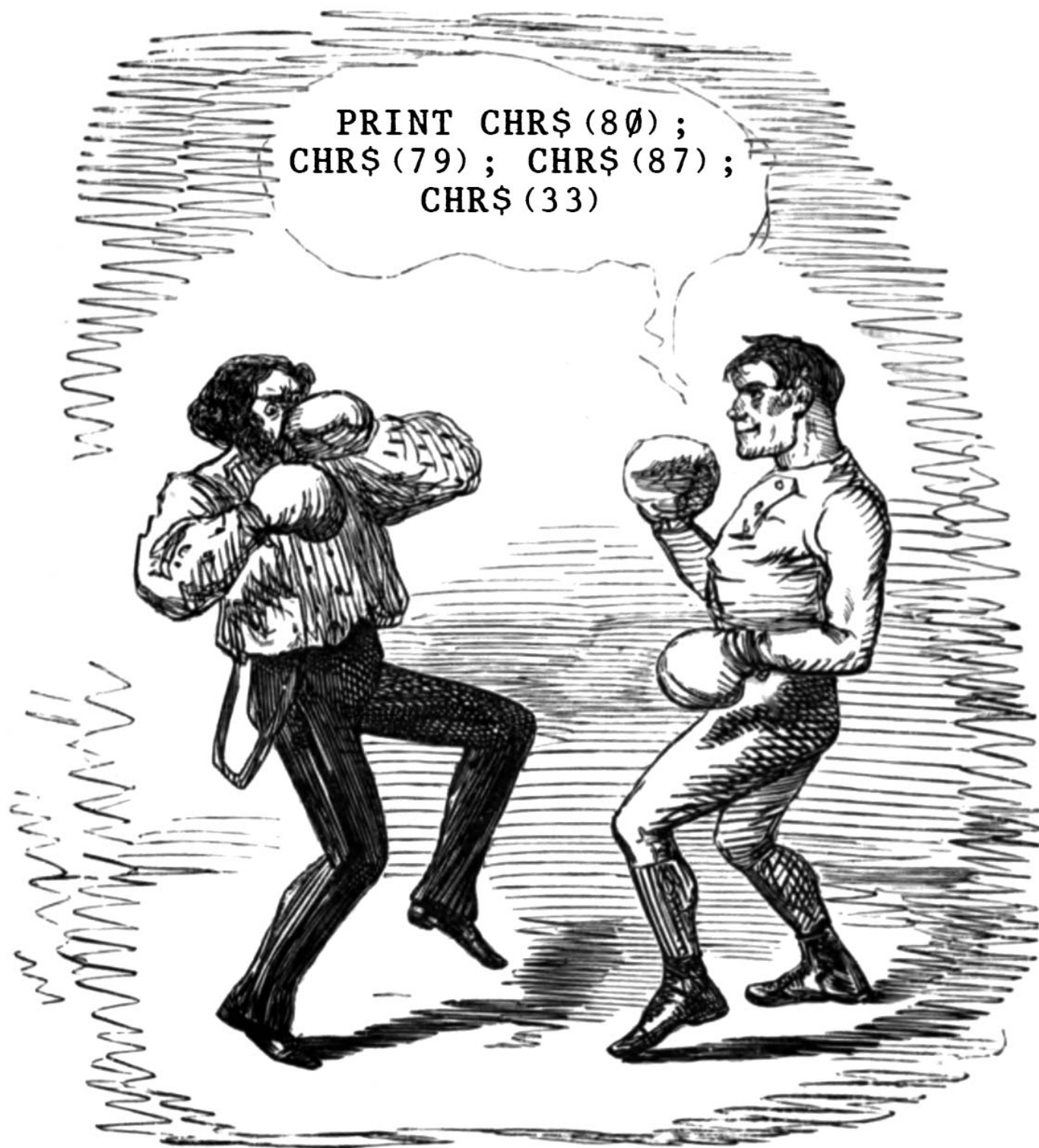
FRONT DOOR METHOD

The process by which a disk is loaded into memory is well defined for normal disks. Certain facts remain true of protected disks regardless of the

method employed. First the disk must contain at least one sector (Track 0, Sector 0) which can be read by the program in the PROM on the disk controller card. Second the code that reads the protected disk must be on the disk. This means that it is possible to trace the boot process by disassembling the code involved in each step of that process. While this can be a formidable task, it is nevertheless theoretically possible to break all protection schemes with this method. The main defense against use of this method is to make it require a great deal of time to accomplish. This could primarily be done in several ways.

One way is to write the code in separate modules or layers. Each layer typically decodes the next layer and recodes the previous layer. It is also vital to verify critical layers to ensure they have not been patched. A second way is to use an interpreted language which introduces an additional level of obscurity and a considerable amount of additional code. Neither of these can be entirely effective, but are important nevertheless.

PRINT CHR\$(80);
CHR\$(79); CHR\$(87);
CHR\$(33)



Appendix C

Nibblizing

This appendix covers in great detail the encoding of data (nibblizing) on the Disk II family of drives (Disk II, IIe, and IIc). Some of this discussion may relate in a general way to encoding techniques on other computers made by Apple. But the details relate specifically to ProDOS and its device driver for a Disk II (or equivalent).

Before starting an explanations of encoding, it is fair to ask why data must be encoded at all? It seems reasonable that the data could simply be written to the disk as it is without any encoding. The reason this can't be done involves the hardware itself. Apple's design of the original Disk II was innovative and used a unique method of recording the data. While this allowed Apple to produce an excellent product, it did require some additional work to be done in software. It is not possible to read all 256 possible byte values from a diskette. This was clearly not an insurmountable problem, but it did require that the data stored on disk be restricted to bytes with certain characteristics.

ENCODING TECHNIQUES

Three different techniques have been used. The first one, which is currently used in Address Fields, involves writing a data byte as two **disk bytes**, one containing the odd bits, and the other containing the even bits. This method is often referred to as “**4 and 4**” encoding, depicting the fact that an 8-bit value is split into two 4-bit pieces. It requires two disk bytes for each byte of data, thus 512 disk bytes would be needed for each 256-byte sector of data. Had this technique been used for sector data, no more than 10 sectors would have fit on a track. This amounts to about 88K of data per diskette, typical for 5 ¼ inch single sided, single density drives.

Fortunately, other techniques for writing data to diskettes were devised that allowed more sectors per track. The earliest technique involved 13 sectors per track. This initial method involved a “**5 and 3**” split of the data bits, versus the “4 and 4” mentioned earlier. Each byte written to the disk contains five valid bits rather than four. This required 410 disk bytes to store a 256-byte sector.

Currently, of course, ProDOS features 16 sectors per track and uses a “6 and 2” split of data bits thereby requiring 342 disk bytes per 256-byte sector. This allows 140K of data per diskette.

The two different encoding techniques (“4 and 4” and “6 and 2”) will now be covered in some detail. The hardware (in order to insure the integrity of the data) imposes a number of restrictions upon how data can be stored and retrieved. It requires that a disk byte have the **high bit set** (the first bit is a “1”), and in addition, it can have **no more than two consecutive zero bits**. Further, each byte can have at most **one pair of consecutive zero bits**.

“4 AND 4” ENCODING

The odd-even “4 and 4” technique meets these requirement — each data byte is represented as two bytes, one containing the even data bits and the other the odd data bits, (shifted one bit right). Figure C.1 illustrates this transformation. It should be noted that the unused bits are all set to “1” to guarantee meeting the two requirements.



Figure C.1: “4 and 4” Encoding Technique

No matter what value the original data byte has, this technique insures that the high bit is set and that there cannot be two consecutive zero bits. The “4 and 4” technique is used to store the information (volume, track, sector, checksum) contained in the Address Field. It is quite easy to decode the data, since the byte with the odd bits is simply shifted left and logically ANDed with the byte containing the even bits. This is illustrated in Figure C.2.

	D ₇	1	D ₅	1	D ₃	1	D ₁	1	(SHIFTED LEFT)
AND	1	D ₆	1	D ₄	1	D ₂	1	D ₀	
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	

Figure C.2: “4 and 4” Decoding Technique

It is important that the least significant bit is a 1 when the odd-bits byte is left shifted. The entire operation is carried out in the device driver for the Disk II.

“6 AND 2” ENCODING

The major difficulty with the above technique is that it takes up a lot of room on the track. Since each disk byte actually contains only four bits of real data, half the bits are wasted. To overcome this deficiency, the “6 and 2” encoding technique was developed. It is so named because, instead of splitting the bytes in half as in the “4 and 4” technique, they are split “6 and 2”. The two bits split off from each byte are grouped together to form additional 6-bit bytes. (They are stored in an area called the Auxiliary Data Buffer.) This means that only two bits are lost in each disk byte. The 6-bit bytes used take the form XXXXXX00 and have values from \$00 to \$FC, each being a multiple of four, for a total of 64 different values. Figure C.3 shows the 6-bit bytes.

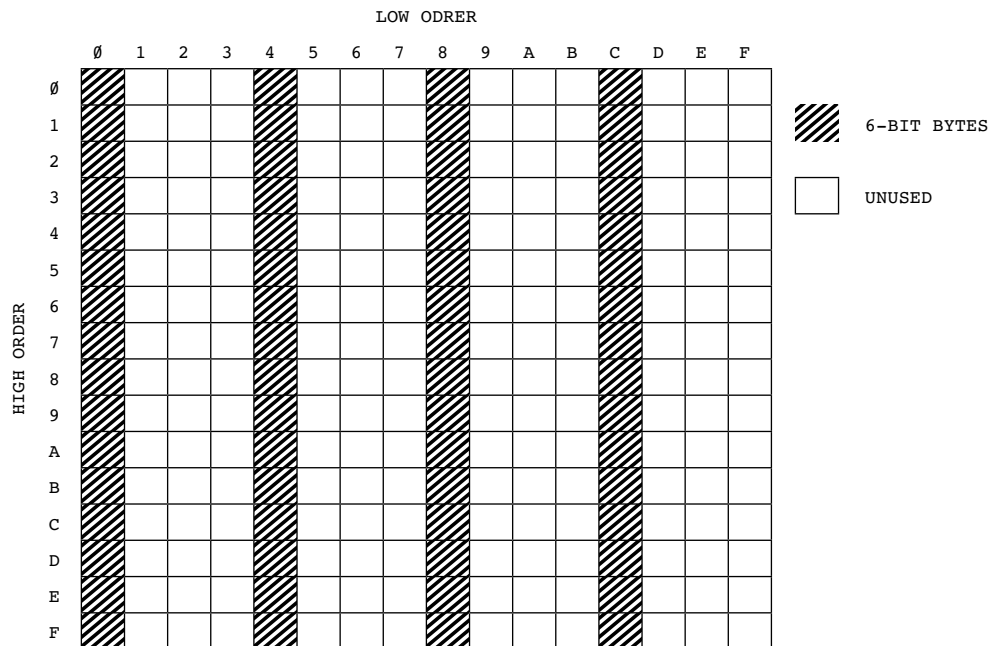


Figure C.3: Valid 6-Bit Bytes

It was necessary to map these 64 6-bit bytes into disk bytes so that they can be stored on the disk. However, there are 72 different bytes ranging in value from \$95 up to \$FF that meet the requirements for valid disk bytes (i.e. the high bit set and one pair of consecutive zero bits at most). After removing the two reserved bytes, \$AA and \$D5, 70 disk bytes remain, and only 64 are needed. An additional requirement was introduced to force the mapping to be one to one, namely, that there must be at least two adjacent bits set, excluding

bit 7. This produces exactly 64 valid disk bytes. A table of valid (and invalid) disk bytes is presented in Figure C.4

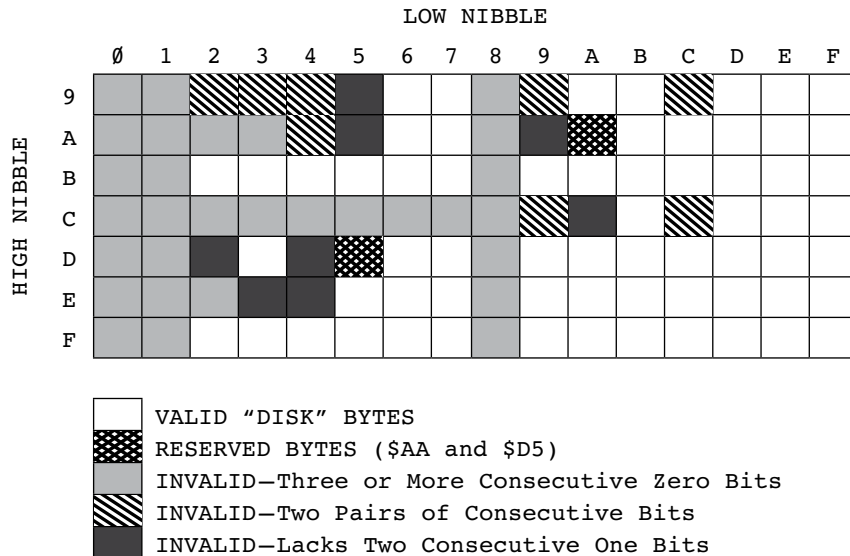


Figure C.4 Valid "Disk Bytes"

The process of converting 8-bit data bytes to disk bytes is a fairly involved process. It has three separate components, two of which we have already mentioned. We will now detail the entire operation required to convert 256 bytes of data into data suitable for diskette storage. An overview of the process is diagrammed in Figure C.5.

THE ENCODING PROCESS

First, the 256 bytes that will make up a sector must be **converted** to 342 6-bit bytes. The number 342 results from finding the total number of bits ($256 \times 8 = 2048$) and dividing by the number of bits per byte ($2048 / 6 = 341.33$). Four of the bits are not used. This operation is done by the "prenibble" routine in the Disk II device driver. The code that performs this operation is fairly involved, as it requires a good deal of bit rearrangement. The results of the operation can however be easily illustrated. Figure C.6 shows how the Auxiliary Data Buffer is formed. The 256-byte User Data Page (containing 8-bit bytes), is passed to the Disk II device driver by ProDOS. Two bits are taken from each byte and put into the Auxiliary Data Buffer. The bits are rearranged slightly during this process. The two bits from each byte are reversed and the order in which they are stored in the Auxiliary Data Buffer is also reversed. The way in which these bits are rearranged and then stored is arbitrary—it could

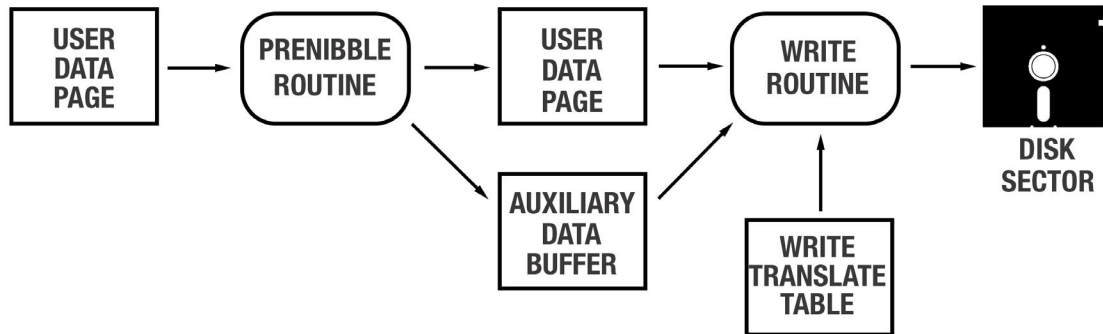


Figure C.5a Writing to the Diskette

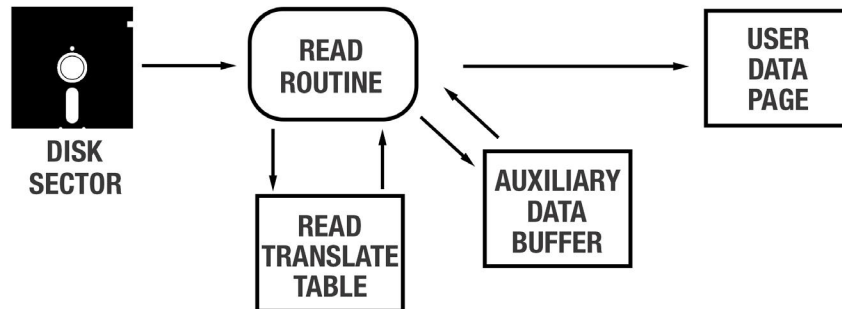


Figure C.5b Reading from the Diskette

have been done differently. The method chosen can be executed rapidly with a small amount of code. The 256-byte User Data Page is in fact unchanged as the bits are copied rather than removed, these bits are ignored (stripped out) when the data is written to the disk. This double usage of the User Data Page eliminates the need for an additional buffer. The Auxiliary Data Buffer contains four areas, one unused and the other three containing segments of the last two bits of the User buffer as is graphically illustrated.

The result of the first step is 342 6-bit bytes. The next step is that of creating a simple checksum that will be used to verify the integrity of the data. Like the Address Field, it also involves **exclusive-ORing** the information, but, due to time constraints during reading bytes, it is implemented differently. The entire block of data is exclusive-ORed with itself offset by one byte. This adds one byte, bringing the block of data to 343 bytes. This process is reversible and, while it cannot aid in recovering damaged data, it does provide a reasonable check on whether the data has been read correctly. The operation of exclusive-ORing the data block with itself is carried out a pair of bytes at a time. This enables the process to be carried out on the fly, that is, while the data is being

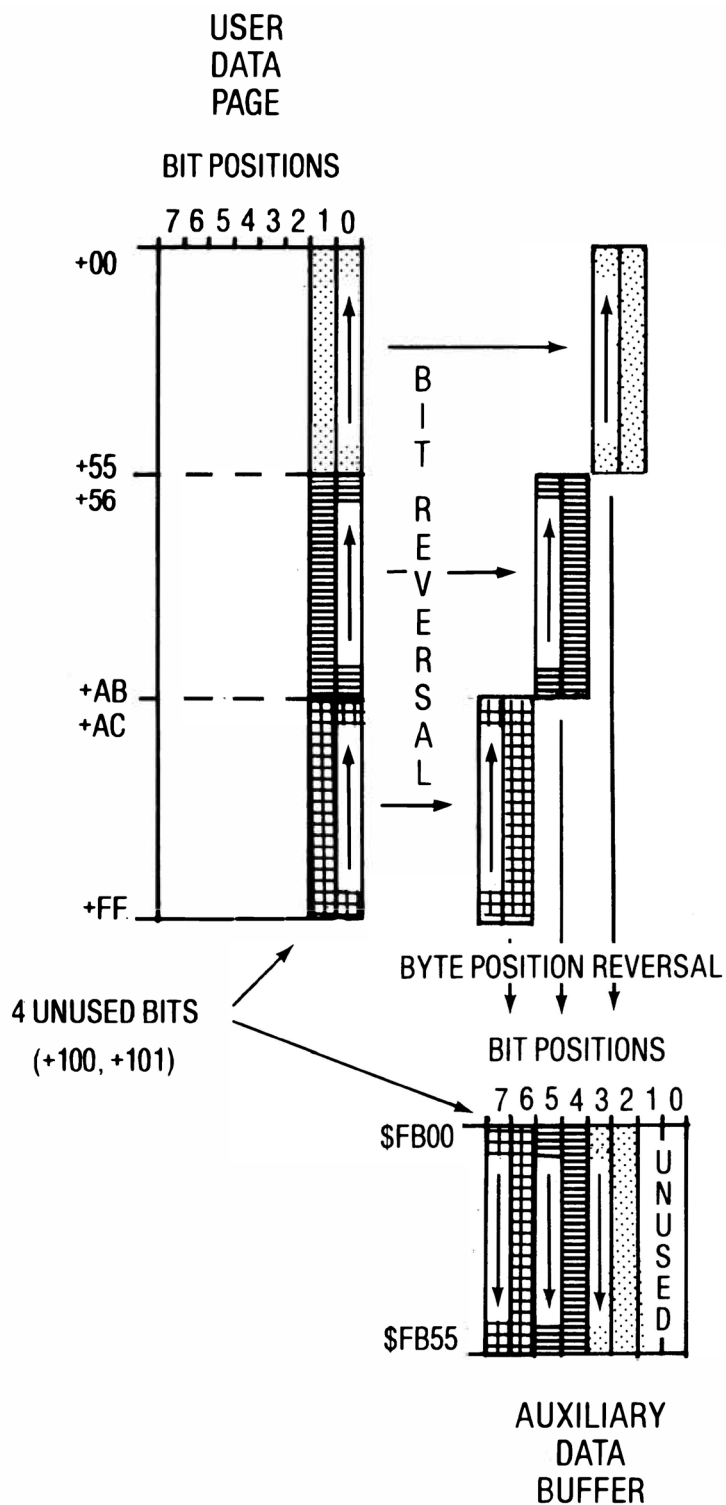


Figure C.6a Forming the Auxiliary Data Buffer

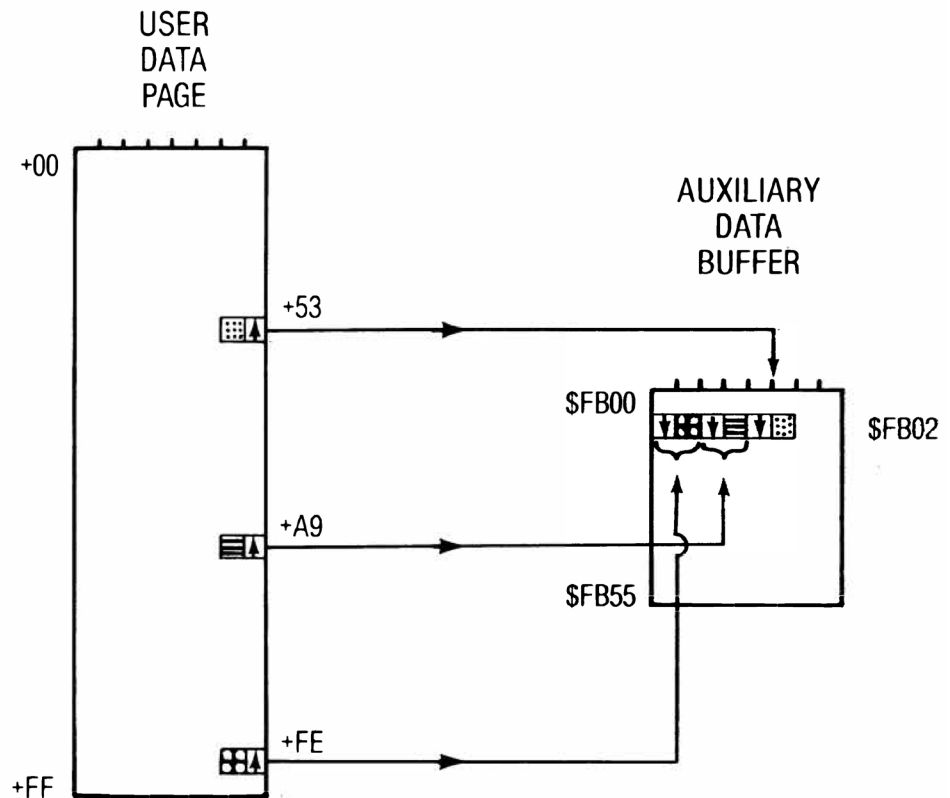


Figure C.6b Forming One Byte of the Auxiliary Data Buffer

read or written. This step and the next are actually done together and are depicted in Figure C.7.

The last step is to **translate** these 343 6-bit bytes to 8-bit disk bytes. This operation is performed using a data table in the Disk II device driver. Figure C.8 shows the mapping of 6-bit bytes to disk bytes in greater detail. Three bytes are highlighted to graphically show how the translation is made. We see for example the \$00 becomes \$96, \$8C becomes \$D3, and \$FC becomes \$FF.

A tabular representation of the same mapping is shown in Figure C.9. It should be noted that this is in fact a **two way mapping**. When bytes are read from the disk they are converted back to 6-bit bytes using this same table.

The reason for this transformation can be better understood by examining how the information is retrieved from the disk. The read routine must read a byte, transform it, and store it—all in under 32 cycles (the time taken to write a byte) or the information will be lost. By using the checksum computation to decode data, the transformation shown in Figure C.10 greatly facilitates the time constraint. As the data is being read from a sector, the accumulator contains the cumulative result of all previous bytes, exclusive-ORed together.

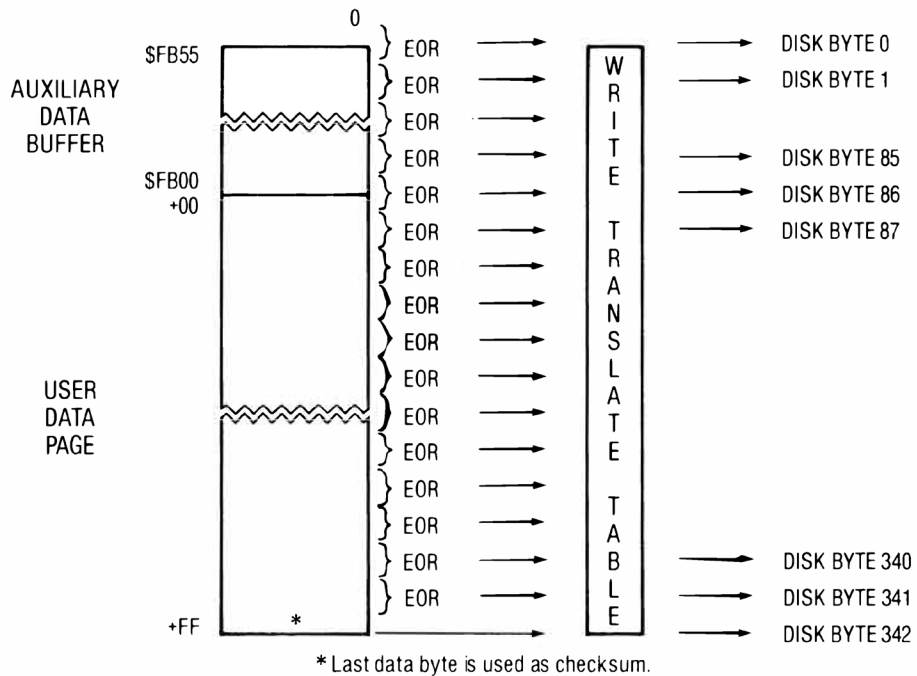


Figure C.7 Writing from Buffers to Disk

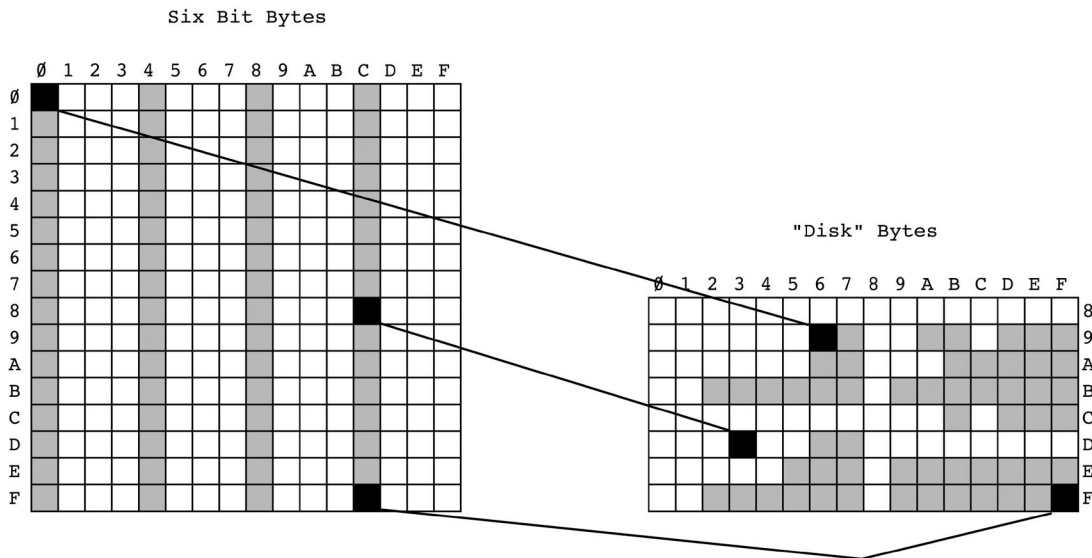


Figure C.8 Relationship of 6-Bit Bytes to Disk Bytes

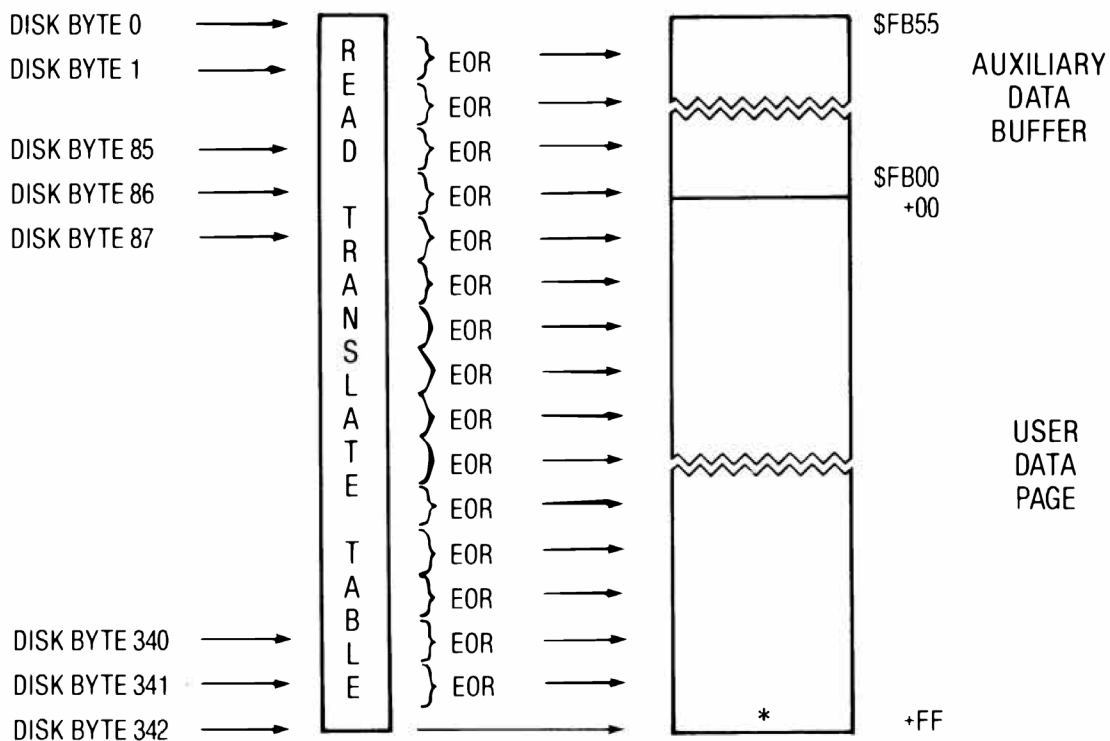
The value of the accumulator after any exclusive-OR operation is the actual data byte for that point in the series. This process is diagrammed in Figure C.10.

00 = 96	10 = B4	20 = D6	30 = ED
01 = 97	11 = B5	21 = D7	31 = EE
02 = 9A	12 = B6	22 = D9	32 = EF
03 = 9B	13 = B7	23 = DA	33 = F2
04 = 9D	14 = B9	24 = DB	34 = F3
05 = 9E	15 = BA	25 = DC	35 = F4
06 = 9F	16 = BB	26 = DD	36 = F5
07 = A6	17 = BC	27 = DE	37 = F6
08 = A7	18 = BD	28 = DF	38 = F7
09 = AB	19 = BE	29 = E5	39 = F9
0A = AC	1A = BF	2A = E6	3A = FA
0B = AD	1B = CB	2B = E7	3B = FB
0C = AE	1C = CD	2C = E9	3C = FC
0D = AF	1D = CE	2D = EA	3D = FD
0E = B2	1E = CF	2E = EB	3E = FE
0F = B3	1F = D3	2F = EC	3F = FF

AA }
D5 } Reserved Bytes

Figure C.9 “6 and 2” Write Translate Table

While containing specific information, the preceding discussion might still be viewed as somewhat of a theoretical presentation. The follow section will show each stage of the transformation that takes place as 256 bytes of data are prepared prior to being written to disk. The data chosen is real data that exists on the ProDOS System disk which will enable the reader to verify the following transformation.



* Last data byte is used as checksum.

Figure C.10 Reading from Disk into the Buffers

STAGE 1

The first stage consists of creating an auxiliary buffer thereby converting the 256 bytes of data to 342 bytes. Each byte in the auxiliary buffer is made up of bits from three different bytes of the original 256-byte data. Please note that the original 256 bytes are still unchanged. Figure C.11 illustrates the results of stage 1, highlighting several bytes to aid in following this process.

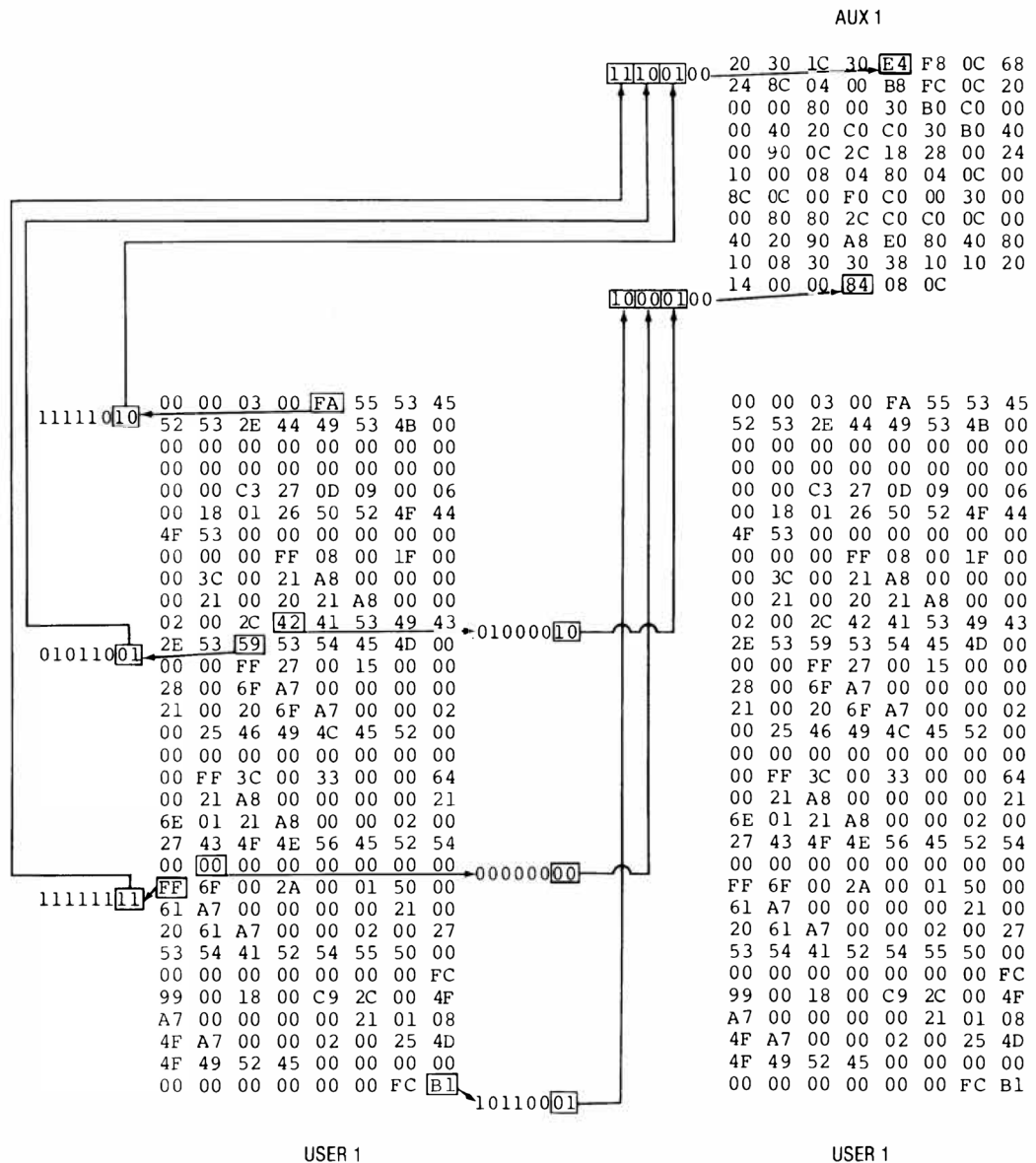


Figure C.11 Example Forming the Auxiliary Data Buffer

STAGE 2

The second stage is to create a checksum by exclusive-ORing the entire 342-byte data block with itself, offset by one byte. If it were not offset, the results would be undesirable (all zeroes). An additional byte is created in this process. While the last byte is in fact unchanged by the process and is independent of the preceding data, it serves as the checksum as seen in Figure C.12.

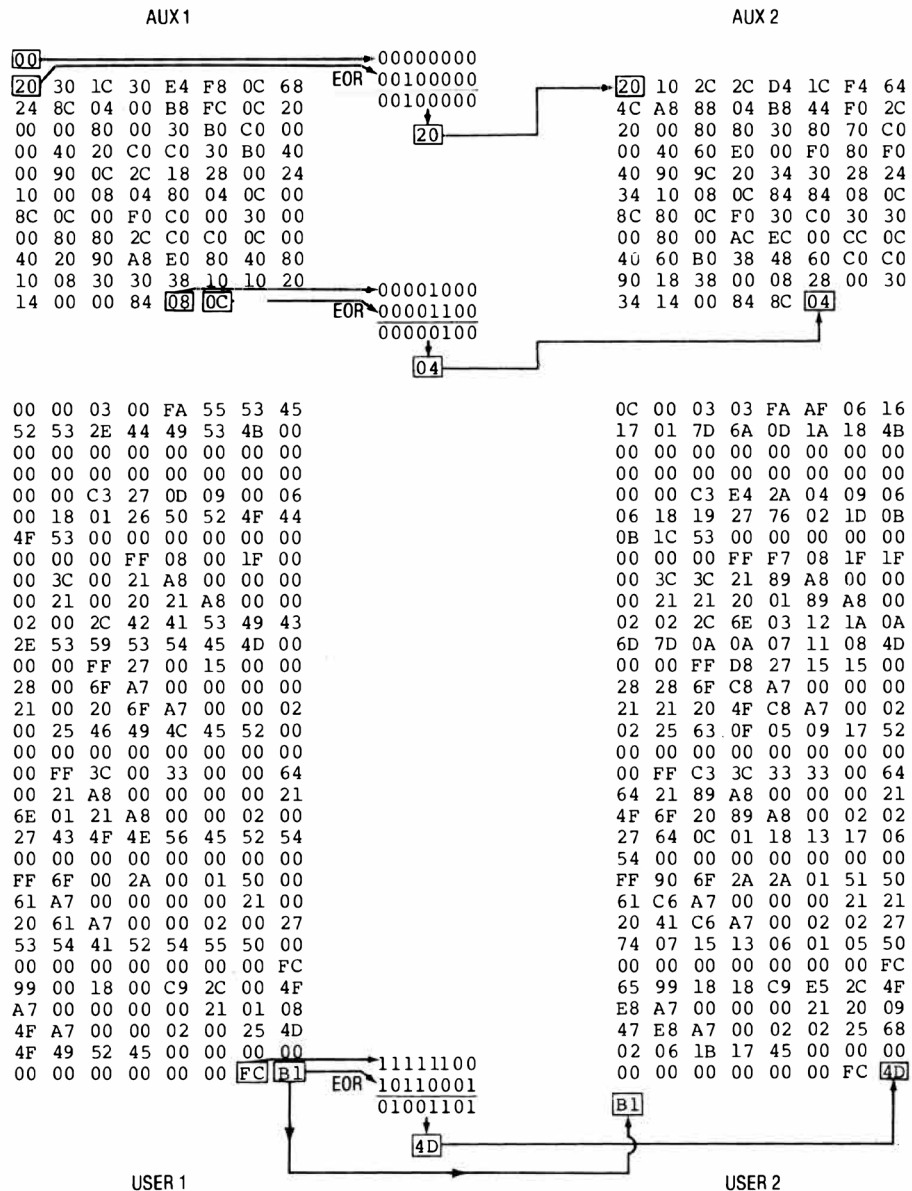


Figure C.12 Example: The Exclusive-ORing Operation

STAGE 3

The third and last stage is to translate the 343 6-bit bytes into disk bytes. This is done with a simple lookup table as shown in Figure C.13. Please note that during this step the last two bits are removed from all bytes before using the table.

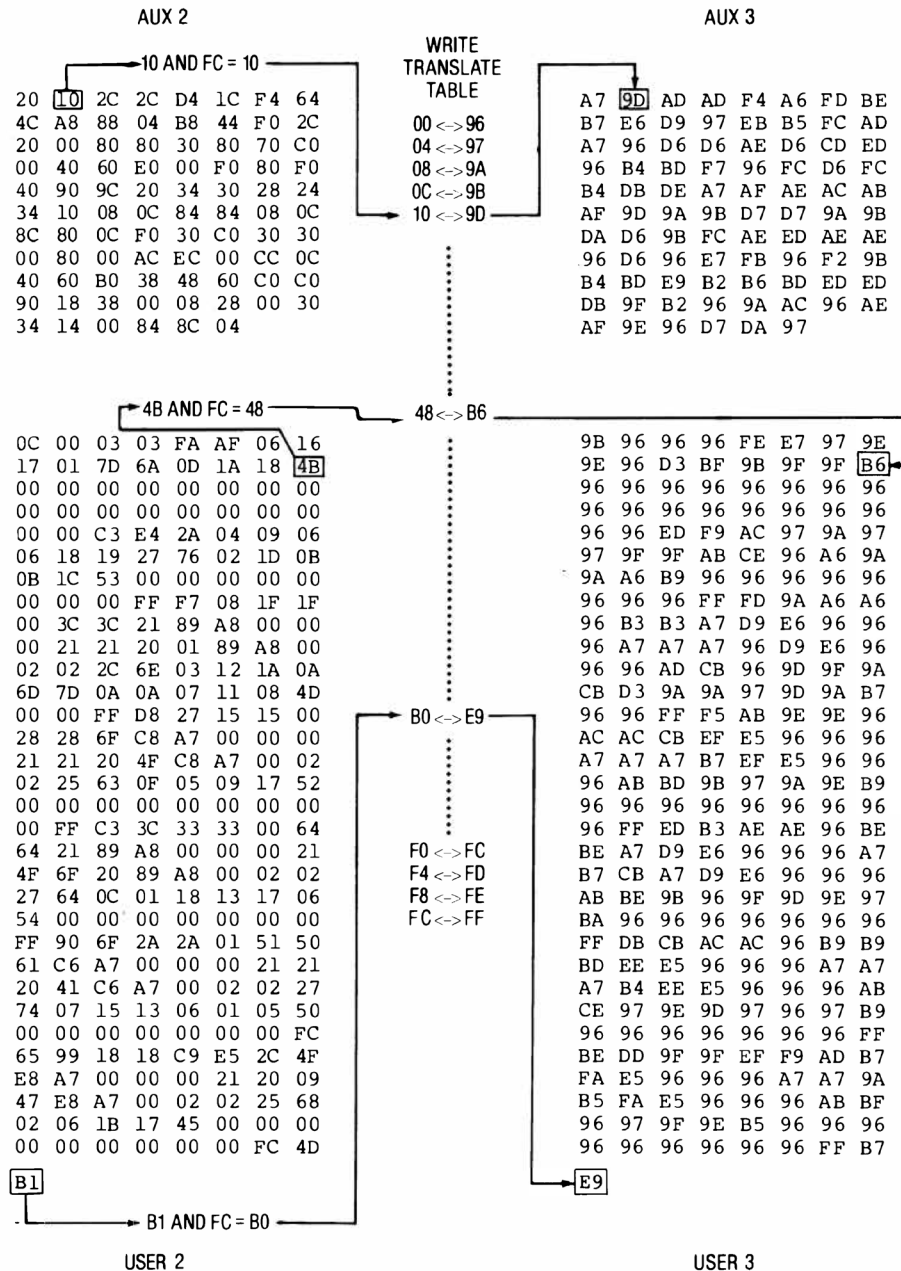


Figure C.13 Example: Translation, the Final Step Before Writing



That's a mighty fine user interface
you got there...

Appendix D

The Logic State Sequencer

Because there is such a close relationship between the disk hardware and the software that controls it, it seems appropriate to examine the firmware that directly responds to the software, that is, the Logic State Sequencer ROM. The code on this ROM actually controls the reading and writing of bits. While the information presented here should enable one to understand the process involved, it is nevertheless intended to be an overview and not a complete analysis.

The Disk II family of drives uses a unique method of storing data on a disk. They use a method named GCR (Group Code Recording) unlike most current disk drives that use FM (Frequency Modulation) or MFM (Modified Frequency Modulation). This enables writing data bytes without the use of clock bits and thereby greatly increases the amount of data that can be stored on a given track. Apple has recently put the Disk Controller Card into a Custom Integrated Circuit. Versions of the Disk Drive Controller Unit (IWM—Integrated Woz/Wendell Machine) are now used on the Apple IIe and the Macintosh. The following discussion is based on the original controller card, but should apply functionally to the new chip as well.

LOGIC STATE SEQUENCER ROM

The Logic State Sequencer is a 256-byte ROM on the disk controller card. The “program” stored there controls the data register, providing the actual means of reading and writing bits. The program on the ROM is unlike traditional software such as BASIC or machine language—it is a simple language with only six different functions or commands available. What makes it different and difficult to follow is how the flow of the program is determined. Traditional languages typically execute instructions in sequence until they encounter a control statement (such as GOTO or GOSUB) that indicates a new location. In the state machine, each byte is both a command (operating on the data register) and a control statement. What is unique is that the location of the next command is only partially determined by the control statement.

The program flow is additionally controlled by four external inputs, two provided by software and two provided by hardware. The software inputs are controlled by four memory locations, \$C08C through \$C08F. The locations are slot dependent (adding the slot number times 16 to the base address gives the appropriate address). Because of the nature of the state machine (timing), this is normally done with the X-register containing the offset (i.e. the slot number times 16). The two inputs provided by the hardware are the presence or absence of a read pulse and the status of the high bit of the data register.

Each of the 256 bytes in the ROM is an available location that can be accessed with the appropriate control statements. Eight bits are needed to indicate all of the locations. Four of these bits are provided by each byte in the ROM and the remaining four bits are provided by the external inputs described earlier. The four bits in the control statement contained in each byte of the ROM indicate what will be called for the next “sequence,” and the four bits from the external inputs indicate what will be called for the next “state.” Figure D.1 depicts the ROM as a two dimensional array, with “sequence” and “state” each providing one dimension of the address of a given element.

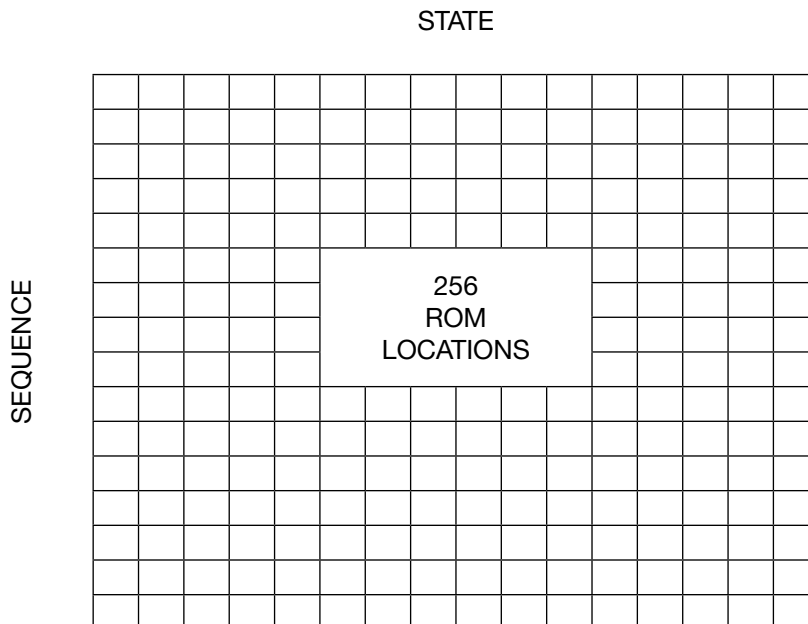


Figure D.1 Sequencer ROM is Addressed by a 4-Bit Input (STATE) and a 4-Bit Control Statement (SEQUENCE)

The 16 sequences are simply the hex numbers 0 through F, and are supplied by the high order nibble of each byte in the ROM. The low order nibble is the command number. For example, the byte “18” would execute command

To make the task easier, the contents of the ROM will be analyzed in four parts, corresponding to the four software states. As mentioned above, the locations \$C08C-C08F (plus the slot number times 16) partially controls the state machine. These four locations control two switches, Q6 and Q7. If one of these addresses is accessed, the appropriate switch will be set as indicated in Table D.2.

ADDRESS	SWITCH	
	Q6	Q7
\$C08C	OFF	—
\$C08D	ON	—
\$C08E	—	OFF
\$C08F	—	ON

Table D.2: State Switches

The first state examined will be with switch Q6 on and Q7 off. This can be described as checking the write protect switch and initializing the state machine for writing. Table D.3 lists the contents of this portion of the state machine ROM. All the instructions are identical (\$0A), each shifting the data register right (command A), bringing in the status of the write protect switch, and then going to sequence 0. This readies the hardware for writing since it is necessary to be in sequence 0 in order to write correctly.

SEQUENCE	HIGH BIT CLEAR		HIGH BIT SET	
	PULSE	NO PULSE	PULSE	NO PULSE
0	0A	0A	0A	0A
1	0A	0A	0A	0A
2	0A	0A	0A	0A
3	0A	0A	0A	0A
4	0A	0A	0A	0A
5	0A	0A	0A	0A
6	0A	0A	0A	0A
7	0A	0A	0A	0A
8	0A	0A	0A	0A
9	0A	0A	0A	0A
A	0A	0A	0A	0A
B	0A	0A	0A	0A
C	0A	0A	0A	0A
D	0A	0A	0A	0A
E	0A	0A	0A	0A
F	0A	0A	0A	0A

Table D.3: State: Q6 On And Q7 Off (Check Write Protect)

The next state examined is with switches Q6 and Q7 off (see Table D.4). This reads data from the disk, shifting in the appropriate bits as a “Pulse” or

“No Pulse” is detected by the hardware. Additionally, once the high bit of the data register is set, the data is retained until a read pulse is detected (0 bits or “No Pulses” are ignored).

SEQUENCE	HIGH BIT CLEAR		HIGH BIT SET	
	PULSE	NO PULSE	PULSE	NO PULSE
Ø	18	18	18	18
1	2D	2D	38	38
2	D8	38	Ø8	28
3	D8	48	48	48
4	D8	58	D8	58
5	D8	68	D8	68
6	D8	78	D8	78
7	D8	88	D8	88
8	D8	98	D8	98
9	D8	29	D8	A8
A	CD	BD	D8	B8
B	D9	59	D8	C8
C	D9	D9	D8	AØ
D	D8	Ø8	E8	E8
E	FD	FD	F8	F8
F	DD	4D	EØ	EØ

Table D.4: State: Q6 Off And Q7 Off (Read)

SEQUENCE	Q6 OFF HIGH BIT		Q6 ON HIGH BIT	
	CLEAR	SET	CLEAR	SET
Ø	18	18	18	18
1	28	28	28	28
2	39	39	3B	3B
3	48	48	48	48
4	58	58	58	58
5	68	68	68	68
6	78	78	78	78
7	Ø8	88	Ø8	88
8	98	98	98	98
9	A8	A8	A8	A8
A	B9	B9	BB	BB
B	C8	C8	C8	C8
C	D8	D8	D8	D8
D	E8	E8	E8	E8
E	F8	F8	F8	F8
F	88	Ø8	88	Ø8

Table D.5: State: Q7 On (Write)

When switch Q7 is turned on (write mode), the presence or absence of a read pulse is ignored. For this reason, the ROM contains two identical 64-byte

sections. Therefore, Table D.5 is presented in slightly different format. Only two operations are carried out, loading the data register from the data bus, and shifting the data out one bit at time, so that it can be written to the disk. Note that only sequence 2 and A carry out any action on the data register.

STEP	DATA REGISTER	READ PULSE	REFER TO TABLE D.4			NEXT SEQUENCE	ACTION**
			COLUMN	SEQUENCE	BYTE		
1	11010101	NO	4	2	28	2	NOP
2	11010101	YES	3	2	08	0	NOP
3	11010101	NO	4	0	18	1	NOP
4	11010101	NO	4	1	38	3	NOP
5	11010101	NO	4	3	48	4	NOP
6	11010101	NO	4	4	58	5	NOP
7	11010101	NO	4	5	68	6	NOP
8	11010101	NO	4	6	78	7	NOP
9	11010101	NO	4	7	88	8	NOP
10	11010101	NO*	4	8	98	9	NOP
11	11010101	NO	4	9	A8	A	NOP
12	11010101	NO	4	A	BB	B	NOP
13	11010101	NO	4	B	C8	C	NOP
14	11010101	NO	4	C	A0	A	CLR
15	00000000	NO	2	A	BD	B	SL1
16	00000001	NO	2	B	59	5	SL0
17	00000010	NO	2	5	68	6	NOP
18	00000010	YES	1	6	D8	D	NOP
19	00000010	NO	2	D	08	0	NOP
20	00000010	NO	2	0	18	1	NOP
21	00000010	NO	2	1	2D	2	SL1
22	00000101	NO	2	2	38	3	NOP
23	00000101	NO	2	3	48	4	NOP
24	00000101	NO	2	4	58	5	NOP
25	00000101	NO	2	5	68	6	NOP
26	00000101	NO*	2	6	78	7	NOP
27	00000101	NO	2	7	88	8	NOP
28	00000101	NO	2	8	98	9	NOP
29	00000101	NO	2	9	29	2	SL0
30	00001010	NO	2	2	38	3	NOP
31	00001010	NO	2	3	38	4	NOP
32	00001010	NO	2	4	58	5	NOP
33	00001010	NO	2	5	68	6	NOP
34	00001010	YES	1	6	D8	D	NOP
35	00001010	NO	2	D	08	0	NOP
36	00001010	NO	2	0	18	1	NOP
37	00001010	NO	2	1	2D	2	SL1
38	00010101	NO	2	2	38	3	NOP

*Normal time to detect a read pulse (if one exists)

**See Table D.1. Notation used here is borrowed from *Understanding the Apple II* by Jim Sather.

Table D.6: A Sequence Example

This discussion should provide a general understanding of the Logic State Sequencer. For a comprehensive look at the disk hardware, an excellent source is *Understanding the Apple II* by Jim Sather, published by Quality Software.

SEQUENCER EXAMPLE

Table D.6 follows the state machine through a number of steps during the read process. It is assumed that a \$D5 has just been read and is now in the data register. The state machine is executing the instruction at column 4 and sequence 2 of Table D.4 and will continue to loop until a read pulse is detected. The instruction being executed is a \$28 which performs a NOP (8 = No OPeration) and remains at sequence 2. In our example, the next byte to read is an \$AA (only the first 5 bits are shown in Table D.6). If the reader can understand this example, it should be possible to construct a similar table for any other read or write example. Note that the column number is controlled by the contents of the MSB of the data register and the presence or absence of a Read Pulse.



That feeling you get when
you try to convert a file
from DOS to ProDOS.

Appendix E

ProDOS, DOS, and SOS

This appendix is intended to assist the reader who is moving programs and data between ProDOS, DOS and SOS operating systems on Apple IIs and Apple IIIs. It is divided into two sections. One deals with the possible problems one might encounter moving from DOS 3.3 or DOS 3.2 to ProDOS with a particular emphasis on the differences in BASIC programming on the two systems. The other section discusses the areas in which ProDOS and SOS are alike, and explains ways in which programs may be written which will run with minimal modifications on either system.

CONVERTING FROM DOS TO PRODOS

The following is a list of potential problem areas when converting programs from DOS 3.3 or DOS 3.2 to ProDOS:

1. Apple DOS allows 30-character file names with embedded special characters and blanks. ProDOS restricts file names to 15 characters. The first must be a letter, and the rest may be letters, numbers or periods. No blanks or other special characters (other than period) may be in a ProDOS file name.
2. The following DOS commands are not supported by ProDOS: **MON**, **NOMON**, **MAXFILES**, **INT**, **FP**, and **INIT**. **MON** and **NOMON** may be entered under ProDOS, but they have no effect.
3. Under ProDOS, the **VERIFY** command does not read through a file to check it for I/O errors. It only verified the file's existence.
4. Although the **V** keyword is syntactically permitted on ProDOS file commands, it is not supported. Programs which depend upon volume numbers must be changed to use volume names instead.
5. When the **APPEND** command is used on a "sparse" random file, it will position at the **EOF** position, not the first "hole."
6. **CHAINing** between BASIC programs is now supported with a command rather than by **BRUNing** a separate file.

7. The most significant bit of each byte is **off** in text files under ProDOS. It is **on** in DOS text files. For example, a blank in DOS was stored as \$A0. Under ProDOS, it is stored as \$20.
8. Under DOS, many programs use statements of the form:
`PRINT CHR$(13);CHR$(4);"dos command to be executed"`
This will not work under ProDOS. The **CHR\$(4)** must be the first item in the **PRINT** list. The **CHR\$(4)** need not be the first thing on an output line, just the first thing in a **PRINT** statement.
9. DOS supports up to 16 simultaneously open files. ProDOS allows only 8.
10. Less memory is available to BASIC programmers under ProDOS. With no files open, the amount of memory available is equivalent to that available under DOS with three open files. Each open file uses 1024 bytes under ProDOS. Under DOS, only 595 bytes are used per file.
11. **HIMEM** should always be set to point to an even page boundary under ProDOS (a multiple of 256).
12. ProDOS does not support Integer BASIC programs.
13. The "**HELLO**" file name must be "**STARTUP**" on ProDOS. DOS allows the user to specify any name for the first file run.
14. All low level assembly language interfaces are drastically different under ProDOS. The MLI must be called to perform disk access wherever the DOS File Manager and RWTS were used in a program. There is no exact equivalent to RWTS in ProDOS, so programs which access the disk by track and sector must be converted to use the **READ_BLOCK** and **WRITE_BLOCK** MLI calls.

WRITING PROGRAMS FOR PRODOS AND SOS

When writing programs which are to run on either ProDOS or SOS, consider the following:

1. ProDOS and SOS volumes are identical in format. Either system can read the other's diskettes.
2. Block 1 on a ProDOS volume contains the SOS boot loader. This program is loaded instead of Block 0 when booted on an Apple III. It searches the directory for SOS.KERNEL and loads it instead of ProDOS. This means that a diskette can be constructed which will boot on either an Apple II or an Apple III.
3. SOS allows up to 16 concurrently open files in BASIC. ProDOS only allows 8.

4. SOS uses different file types than ProDOS. A ProDOS **CATALOG** on a SOS diskette will produce hex codes for file type but this is normal. Table E.1 shows all ProDOS and SOS file types currently defined.
5. SOS memory management allows programs to allocate and free segments of memory by making system calls. Under ProDOS, programs must manage memory themselves by marking pages free or in use in the System Global Page.
6. SOS system calls are, for the most part, very similar to ProDOS MLI calls. The areas in which differences occur are: ProDOS filing calls apply only to block devices (disks), but SOS filing calls apply to all devices; **GET_FILE_INFO** under SOS gives the **EOF** position of a file, whereas ProDOS's **GET_FILE_INFO** does not; SOS's **SET_MARK** and **SET_EOF** positions may be given as relative to the current position, but ProDOS requires them to be absolute.
7. SOS interrupts are prioritized and managed by device drivers; however, ProDOS interrupts are polled sequentially and are managed by interrupt handlers installed using MLI calls.

Table E.1: ProDOS and SOS File Types

HEX TYPE	PRODOS NAME	OS	MEANING
\$00		both	Typeless file
\$01		both	Bad blocks file
\$02		SOS	PASCAL code file
\$03		SOS	PASCAL text file
\$04	TXT	both	ASCII text file
\$05		SOS	PASCAL text file
\$06	BIN	both	Binary File
\$07		SOS	Font file
\$08		SOS	Graphics screen file
\$09		SOS	Business BASIC program file
\$0A		SOS	Business BASIC data file
\$0B		SOS	Word processor file
\$0C		SOS	SOS System file
\$0D-\$0E		SOS	SOS reserved for future use
\$0F	DIR	both	Directory file
\$10		SOS	RPS data file
\$11		SOS	RPS index file
\$12-\$18		SOS	SOS reserved for future use
\$19	ADB	ProDOS	AppleWorks data base file
\$1A	AWP	ProDOS	AppleWorks word processing file
\$1B	ASP	ProDOS	AppleWorks spreadsheet file
\$1C-\$BF		SOS	SOS reserved for future use
\$C0-\$EE		ProDOS	ProDOS reserved for future use
\$EF	PAS	ProDOS	ProDOS PASCAL file
\$F0	CMD	ProDOS	Added command file
\$F1-\$F8		ProDOS	ProDOS user defined file types
\$F9		ProDOS	ProDOS reserved for future use
\$FA	INT	ProDOS	Integer BASIC program file
\$FB	IVR	ProDOS	Integer BASIC variables file
\$FC	BAS	ProDOS	Applesoft BASIC program file
\$FD	VAR	ProDOS	Applesoft BASIC variables file
\$FE	REL	ProDOS	EDASM relocatable object module file
\$FF	SYS	ProDOS	System file

Appendix F

Differences Between the ProDOS 8 Versions

This Appendix identifies the changes to ProDOS 8 that were introduced with the 1.2 and 1.3 versions. Although we believe this to be a fairly thorough list, there may have been a few changes (especially deletions) that we didn't catch. Quite a few changes that were apparently made only to save space are not listed here.

CHANGES INTRODUCED IN THE 1.2 VERSION

The changes that were made to the 1.1.1 version of ProDOS to produce ProDOS 8, Version 1.2, are listed below. Addresses given here are Version 1.2 addresses.

RELOCATOR

1. When running on a IIGS with ProDOS 16 installed, ProDOS 8 is entered from PQUIT (the ProDOS 16 Quit Handler). In this case the Relocator is entered at \$2003 instead of \$2000 [2000-2005].
2. The ProDOS 8 version number is now stored in the MLI subdirectory header data area [2048-204C].
3. Always checks to see if running on a IIGS, and if so sets a flag (2278). Also sets E100BD=0 if ProDOS 8 is the initial boot on a IIGS. [2079-2680].
4. Sets aux stack pointer to \$FF [20E6-20F5, 2111-2115].
5. If operating on a IIGS, skips logic that searches for slot 3 80-column card [212A-2139].
6. If operating on a IIGS, installs the IIGS Clock Code [21AD-21D0, 22D3-22DA].
7. Now checks for an AppleTalk Initialization File (ATINIT file) before looking for a .SYSTEM file. If the ATINIT file is found, it is loaded and executed, then the search for a .SYSTEM file commences. [22DB-2381].

8. The list of devices is now ordered differently. It recognizes the SmartPort and allows four Slot 5 SmartPort units to be accessed as Slot 5, Drives 1 and 2 and Slot 2, Drives 1 and 2. If Slot 2 is being used by a storage device, however, only the first two devices on the Slot 5 SmartPort can be accessed. It also changes the search order, making sure that Disk II devices are searched last when a device scan takes place (such as during an MLI ON_LINE call). [2668-271B, 275D-2767, 2814-28AA].
9. A bug in the /RAM driver (which we pointed out in the 1.1.1 supplement) that allowed a block read of block 7 (which doesn't exist) has been corrected [2D4B].
10. The /RAM caller, which operates in high RAM, now contains a \$60 (RTS instruction) at address \$FF58. Peripheral cards sometimes call that address to figure out which slot they are in, and in case they forget to set ROM for reading, the call will still work. [2E56-2E58].
11. A subroutine that sets high RAM for reading/writing was created to save space in the code [2518-251E].

MLI AND MI GLOBAL PAGE

1. The Global Page now pushes the P-Register and disables interrupts before calling the actual MLI [DE01-DE04, DE1C-DE21, BF4B-BF4F].
2. Setting the MLIACTIVE flag a little differently now allows nested calls to the MLI by interrupt routines [DEBF-DE91].
3. A new MLI command was introduced (command=\$82), that allows the user to install a routine to handle unclaimed interrupts [DEFF-DF0B, FD23-FD2C].
4. If there is no unclaimed interrupt handler, ProDOS 8 now counts unclaimed interrupts, and will allow 255 of them to occur before finally issuing a fatal error. This allows a brief time for the unknown interrupt to stop interrupting. [DFB3-DFB7].
5. If operating on a IIGS and system death occurs, the NEWVIDEO softswitch is set to 0, reinitializing the IIGS video [E013-E016].
6. Processing for the ON_LINE command now frees the VCB entry for a device that was previously on-line but has been taken off-line [E28A-E2A4].
7. A subroutine that reads a block where the block number is in the A and X registers was added to save space in the code [EBC9-EBD0].

8. The error message that results when a file being opened has an illegal storage type has been changed from “incompatible directory format” to “unsupported storage type” [EEC7–EECA].
9. An error type \$C is now indicated when truncating or deleting a file and the file’s storage type is illegal [FA4D–FA4E].
10. To save space in the QuitCode Caller, some in-line code was changed to a loop [FCAF–FCB7, FCD8–FCE0].

QUIT CODE

1. Uses standard character set instead of alternate character set [1006–1008].
2. Sets normal 40-column screen in a safer way, such that screen hole values are preserved [100C–1011].
3. Message display routine is modified [1033–1034, 11D6–11D7, etc.].
4. The method of displaying the current prefix is changed so that it is always written to the same screen location [104D–105C].
5. User can now backspace with the DELETE key as well as left arrow [107C–107F, 10FC–10FF].
6. The method of inputting the Application name was modified [10E7–10E9].

CLOCK CODE

1. The code for the ThunderClock includes a lookup table to determine the year based on the day of the year and the day of the week. This table is only good for a span of five or six years. The table released with Version 1.1.1 was good for the years 1982-1987. The table released with Version 1.2 covers the years 1986-1991. [D7B8–D7BE].
2. A completely separate clock routine is provided in Version 1.2 in case ProDOS 8 is operating on a IIGS. If so, the IIGS Clock Code is always enabled. It is written in 65816 and calls the ReadTimeHex tool in the tool kit to read the clock. [D742–D790].

CHANGES INTRODUCED IN THE 1.3 VERSION

The changes that were made to the 1.2 version of ProDOS 8 to produce ProDOS 8, Version 1.3, are listed below. Addresses given here are Version 1.3 addresses.

RELOCATOR

1. The boot message now includes a line that says “ALL RIGHTS RESERVED.” Chalk up one for the legal department! [25F6–2671].
2. A ProDOS Status call now immediately precedes the SmartPort Status Call. This is because the SmartPort interface does not set up its device list until it receives a ProDOS Status call. Earlier versions of ProDOS 8 may not always find all SmartPort devices. [286E–2894].

MLI

1. When files are deleted, previous versions of ProDOS zero out all but the first block of discarded index blocks. Now such index blocks will not be zeroed, but the pages of these blocks will be flipped. That is, the high byte of the block numbers will be exchanged with the low byte of the block numbers. [F992–F99A, FBC7–FBDC].
2. Previous versions of ProDOS forgot, in certain cases, to rewrite index blocks that were being discarded when shortening or deleting files. Now such blocks will always be rewritten to disk in a zeroed (shortened file) or flipped (deleted file) form. [FAB8–FABC, FB91–FB93].
3. A poorly-written loop in the QuitCode Caller that was added for Version 1.2 was rewritten for Version 1.3. The Version 1.2 code might cause problems on a IIGS. [FD05–FD0C].

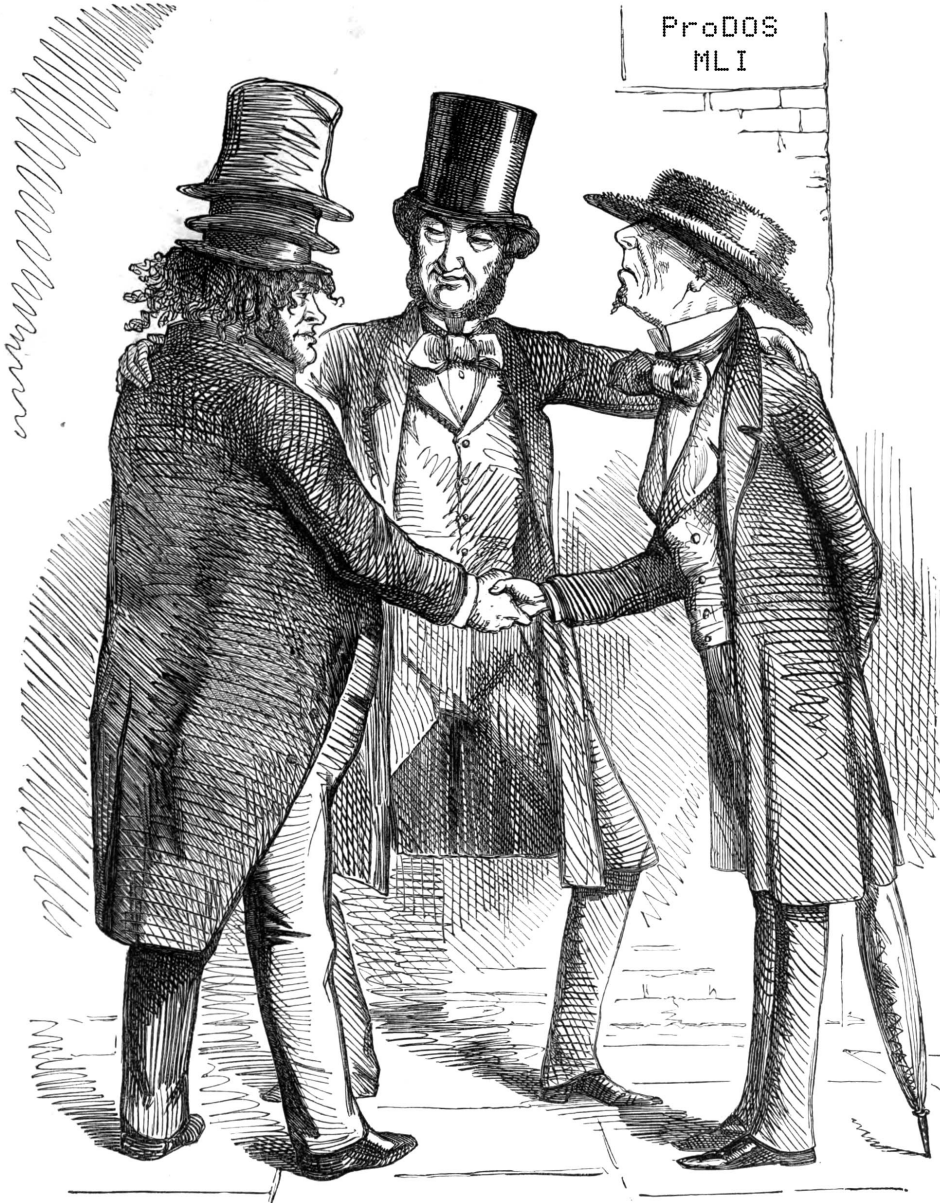
DISK II DEVICE DRIVER

1. There is a routine in the Disk II Device Driver that clears phases in case the Disk II device is sharing transmission lines with SmartPort devices. This routine was patched in Version 1.3 so that phases are now cleared with LDA instructions instead of STA instructions. This eliminates bus fights that can, in some situations, cause unwanted writing to the floppy disk. [D6C3–D6CE].

BUGS IN VERSIONS 1.2 AND 1.3

It is fair to say that both Versions 1.2 and 1.3 of ProDOS 8 are relatively bug free. Perhaps a few escaped our notice, but we know of only three minor bugs, which are as follows:

- MLI, Versions 1.2 and 1.3, at EC64. This bug has been in ProDOS since day 1. Although there is no easy way to correct the problem (because a three-byte instruction is needed where there are only two bytes), any serious problems can be avoided by putting NOP's at EC64 and EC65 (3D64 and 3D65 in load location). This bug can only take effect when a storage type 0 is found (not likely unless disk swapping) and a lot of files are open simultaneously.
- MLI, Version 1.2 only, at FCD8. A loop that indexes around the 64K boundary may cause problems on a IIGS. Use version 1.3 or recode the loop so that the boundary crossover is eliminated.
- MLI, Version 1.3 only, at FBCD. A 65C02 instruction snuck into the code, which will be disastrous when Version 1.3 is run on a computer with a 6502 processor (Apple II+, unenhanced IIe). It is easily patched: just change the byte at FBCD (4CCD in the load image) from \$80 to \$F0, because BEQ works fine here.



Adding New Commands to ProDOS

Miscellanea



Could you spare a few bits?



SETUP.SYSTEM

BY SEAN NOLAN

This article describes a program which lets you easily install customizing routines, such as a /RAM disk and clock card drivers, on bootup into ProDOS. It mimics, under ProDOS 8, the **SYSTEM.SETUP** feature of ProDOS 16. It is emphatically placed in the public domain: I urge software publishers to take it and include it where appropriate with their own programs. I only ask that you take the program unmodified. It would be unfortunate if variants of the program spread around, each subtly incompatible with the others.

One feature which makes ProDOS 16 a more powerful operating system than ProDOS 8 is its flexible and complicated startup sequence. This includes the ability to boot up into either ProDOS 8 or ProDOS 16 based applications, as well as automatically installing, on bootup, patches in /RAM to the Apple II GS toolbox. ProDOS 16 accomplishes this by loading and executing all files found in the /SYSTEM/SYSTEM.SETUP subdirectory. These files must include **TOOLSETUP**, but may include other files for installing desk accessories, diagnostic and debugging routines, or anything else you can think of.

SYSTEM.SETUP is a nice idea. It allows you to specify what happens at boot up simply by placing the correct files in a particular subdirectory. Unfortunately, ProDOS 8 does not support **SYSTEM.SETUP**. If it did, it would be much easier to install IIe /RAM disk drivers, clock card drivers, and II GS Classic Desk Accessories on ProDOS 8 disks.

Suppose you have a large memory card in your IIe, and you want a /RAM disk driver for it to be installed automatically every time you boot. Here are two ways you could do it:

1. If the /RAM disk driver is a system program, you could place it first on the disk so that it's automatically called on boot up. When the system program quits, you then type in the name of the next system program to run, such as **BASIC.SYSTEM**.

2. If the /RAM disk driver is a binary file, you can place **BASIC.SYSTEM** first, and have a BASIC program called **STARTUP** which **BRUNs** the /RAM disk driver. This is fine, as long as you plan to boot up into BASIC.

If you have both a large memory card and a clock card, you may need to install drivers for both of them. Now things become more difficult for you, particularly if your memory card came with a **SYStem** file program to install the /RAM disk, and your clock card came with a **BiNary** file clock driver. **SETUP.SYSTEM** (note: *not* **SYSTEM.SETUP**) automates the bootup sequence for you. To use it, assemble it and make it the first **SYStem** program on your disk whose name ends in “**.SYSTEM**”. This ensures that it will be run on boot up. Then create a subdirectory called **SETUPS**, containing the files which you want to be executed on boot up. That’s all there is to it.

Many existing programs already work with **SETUP.SYSTEM**. I have used it with the RAM disk drivers for both Applied Engineering’s and Checkmate Technologies’ Ile memory cards, and with *Diversi-Cache* and *Diversi-Hack* (1165 classic desk accessories by Bill Basham).

When run, **SETUP.SYSTEM** will search for the **SETUPS** subdirectory in the volume directory of the last disk accessed. It then loads and executes, in turn, each **SYStem** or **BiNary** file in the **SETUPS** subdirectory. These files I refer to as “setup files.” When all setup files have been called, **SETUP.SYSTEM** then looks for the second system file in the main volume directory whose name ends in “**.SYSTEM**”, and runs that. This second “**.SYSTEM**” program is your main application. It can be **BASIC.SYSTEM**, your assembler, word processor, *AppleWorks*, or any other ProDOS application.

SETUP.SYSTEM was especially written to be compatible with already existing /RAM disk drivers, clock drivers, and Apple IIGS Classic Desk Accessories under ProDOS 8. The rules for compatibility are simple. Setup files must be either **BiNary** files or **SYStem** programs in the **SETUPS** subdirectory. This is in contrast to ProDOS 16’s **SYSTEM.SETUP**, which uses new ProDOS 16 file types exclusively. You may place other file types in the **SETUPS** subdirectory, such as text files, but **SETUP.SYSTEM** will ignore them. **BiNary** setup files may load anywhere in either memory range **\$2000-\$3EFF** or **\$8000-\$BEFF**. They should exit with an **RTS**. You may also place **SYStem** programs in the **SETUPS** subdirectory. These **SYStem** programs, as usual, must load at **\$2000**, and must not exceed **\$B8FF** when loaded. **SYStem** setup programs should exit with a ProDOS **QUIT** call. Most **SYStem** programs do this, although there are exceptions. A few exit by looking for the next **SYStem** program to run by name, or look for and run the second “**.SYSTEM**” file on

the disk. Programs which do this are incompatible with **SETUP.SYSTEM**. Among the few system programs which are incompatible for this reason is the original version of the ProDOS **FILER**.

Once the setup program is in memory, it may use any memory except for pages **\$BD** and **\$BE**, which is where **SETUP.SYSTEM** lives in hibernation, and the ProDOS global page at **\$BF00**. Because of this memory restriction, some existing **SYStem** programs may not be used as setup files. You should not, for instance, place **BASIC.SYSTEM** in the **SETUPS** subdirectory. **BASIC.SYSTEM** relocated itself to **\$9A00-\$BEFF**, overwriting **SETUP.SYSTEM**. If you do so, and then type “**BYE**” from **BASIC**, your Apple will explode into a reball, destroying all life other than insects within a half mile radius. Really.

A third reason why system programs may not work in the **SETUPS** subdirectory is that they may require subsidiary files—which must also be placed in the **SETUPS** subdirectory. If these subsidiary files are **BiNary** or **SYStem** files, **SETUP.SYSTEM** will eventually try to call them on their own. This problem happened to me when I moved the Apple IIGS *System Utilities* into my **SETUPS** directory. (Why? So I could copy some files onto my **/RAM** disk on bootup.) The problem was solved by changing the file type of the three **BiNary** files which the *System Utilities* needed. I gave them file type codes of **\$03**, which is a defunct Apple III Pascal Text file code. The point was, **SETUP.SYSTEM** cared what the file types were and the *System Utilities* didn't. I should point out that placing random system programs in your **SETUPS** subdirectory can be dangerous. By trial and error, I found that the GS *System Utilities* left the crucial pages **\$BD** and **\$BE** undamaged.

If you plan to write new setup programs, I recommend that you make them **BiNary** files rather than **SYStem** programs. **SETUP.SYSTEM** temporarily redirects the ProDOS quit vector at **\$BF03** into itself before calling any **SYStem** programs in the **SETUPS** subdirectory. Therefore, when the setup program executes a ProDOS **QUIT** call, control is returned to my code. This is not kosher programming, and may not work with future revisions of ProDOS. I added the feature solely so that many already existing **SYStem** programs could be used with **SETUP.SYSTEM**.

TYPING IT IN

Although the program below was assembled using the Merlin Pro assembler, I tried to avoid as many Merlin-specific psuedo ops as possible and present a generic assembly listing. In order to assemble it on other assemblers, some minor changes will be necessary: The psuedo-op **ASC** in Merlin creates either

high or low ASCII characters, depending on whether the string is surrounded by a single quote (low ASCII) or a double quote (high ASCII), I used both varieties. Users of EDASM, APW, or ORCA/M should add the **MSB** psuedo-op where needed. The last byte of the program is a checksum byte. It is the exclusive-OR of all the previous bytes in the program. If you assemble the program using Merlin, be sure to check that the byte you get is the same as the one in the listing. If it isn't, you made a typo.

If you are not using Merlin, replace the last line with

DFB \$E8

and run the following program to check that the code is correct:

```
10 PRINT CHR$(4) "BLOAD SETUP.SYSTEM, A$2000, TSYS"
20 FOR X = 8192 TO 8703
30 SUM = SUM + PEEK(X)
40 NEXT
50 IF SUM = 64732 THEN PRINT "OK" : END
60 PRINT "ERROR"
```

PROGRAM LOGIC

SETUP.SYSTEM is not a model of good programming practice. In order to get all the functionality I wanted into 512 bytes, I had to resort to self-modifying code. If the program went one byte over, to 513 bytes, the storage space on disk would mushroom from one block to three.

Among the first tasks of **SETUP.SYSTEM** is to relocate itself from it's load address of **\$2000** to it's running address of **\$BD00**. **SETUP.SYSTEM** must move itself out of the way so that it can load other system programs without overwriting itself. Lines 48 through 54 in the program listing accomplish this. Next, the program (lines 61 through 72) tries to determine the name of the disk from which **SETUP.SYSTEM** was run. This name is saved so that the program can set the ProDOS prefix to the volume directory at a later time. The quit vector in the ProDOS global page is then saved (lines 73-76) so that it can be restored on exit.

Control returns to line 78 at the label **MainLoop**, before and after executing each of the setup files. The reset vector and the quit vector are both redirected to point to **MainLoop**, to increase the chances of **SETUP.SYSTEM** regaining control after calling setup programs. The system bitmap is then cleared so that we can load the setup file (lines 87-95), and the machine is then put into as normal a state as possible (lines 96-103).

Lines 106 through 127 make sure that the original volume is in the disk drive, and sets the ProDOS prefix to that volume. If the disk isn't around, a warning message is displayed, and the user is given a chance to insert the correct disk.

Line 129 calls a subroutine called **NextFile**, which reads the **SETUPS** subdirectory until it finds the name of the next **BIN**ary or **SY**stem program. It leaves that name in a known place. If it finds no more **SY**stem or **BIN**ary files it exits with the carry set. This will happen when all of the setup files have been loaded and executed—that case will be dealt with later.

If **NextFile** found the name of a setup file, we proceed to line 132. Here the prefix is set to the **SETUPS** subdirectory, and the setup file is loaded into memory by calling the subroutine **ReadFile**, which starts at line 240. If there was no error when loading it, the setup file in memory is called. One way or another, control returns to **MainLoop**, either through the ProDOS **QUIT** call, the user pressing **RESET**, or the setup file exiting with an **RTS**.

Once all the setup files have been loaded, we go to line 139 and restore the original ProDOS quit vector. Lines 145 through 149 then modify the **NextFile** routine so that it works rather differently. Where before **NextFile** looked for either **BIN**ary or **SY**stem files, now it will look for only **SY**stem ones. **NextFile** used to look in the **SETUPS** subdirectory. It now looks in the volume directory. **NextFile** will now be used to find the name of the system program to exit to.

NextFile is called repeatedly, returning the names of all **SY**stem programs in the volume directory (line 150). Lines 152-159 make sure that the filename ends in “**.SYSTEM**”. If it passes this test, we then check that it is not the first “**.SYSTEM**” file on the disk (lines 160-162). The first “**.SYSTEM**” file on the disk is presumably “**SETUP.SYSTEM**”, and we don't want to run that, or we'd go into an endless loop. The system file is loaded, and called (line 163) with the same subroutine, **ReadFile**, which was used to load and call setup files. The difference is that this time we never come back, since the ProDOS quit vector has been restored to its original value.

```

1  *
2  * SETUP.SYSTEM by Sean Nolan
3  *
4  * A Proposed Startup File Standard
5  *
6  * Published in Call-APPLE, November, 1987
7  * This program is in the public domain.
8  *
9  * This program mimics the ProDOS 16
10 * SYSTEM.SETUP convention. It can be used
11 * to install RAM disk drivers, clock
12 * drivers, and IIGS Classic Desk
13 * Accessories on bootup under ProDOS 8.
14 *
15 * This program loads and calls all BINary
16 * and SYSTem files in a subdirectory named
17 * SETUPS. It then looks for the second
18 * system program in the volume directory
19 * whose name ends in ".SYSTEM", and runs
20 * that.
21 *
22 *
23          TYP  $FF          ;save as a system file
24          ORG  $BD00        ;load at $2000, but run at $BD00
25 ***** equates
26 CH      =      $24
27 IN2     =      $280
28 FILETYPE =      IN2+16
29 AUXCODE =      IN2+31
30 RESET   =      $3F2
31 IOBUFFER = $B900
32 PRODOS  = $BF00
33 QUITVECT = $BF03
34 DEVNUM  = $BF30
35 BITMAP  = $BF58
36 INIT    = $FB2F
37 VTABZ   = $FC24
38 HOME    = $FC58
39 RDKEY   = $FD0C
40 SETVID  = $FE93
41 SETKBD  = $FE89
42 SETNORM = $FE84
43 ***** boot code
44 VOLNAME =      *          ;The first 17 bytes are overwritten with the
45                               ;name of the volume from which this was run.
46          LDX  #1          ;mark page $BD as free in the system bitmap
BD00: A2 01 47          STX  BITMAP+23 ;so we can put Online result in our code.
BD02: 8E 6F BF 48          DEX
BD05: CA 49          LOOP1 LDA  $2000,X ;relocate this program to $BD00-BEFF
BD06: BD 00 20 50          STA  $BD00,X
BD09: 9D 00 BD 51          LDA  $2100,X
BD0C: BD 00 21 52          STA  $BE00,X
BD0F: 9D 00 BE 53          INX
BD12: E8 54          BNE  LOOP1
BD13: D0 F1 55          DEX
BD15: CA 56          TXS          ;init stack pointer
BD16: 9A 57          JMP  ENTER    ;jump to relocated code
BD17: 4C 21 BD 58          DIRNAME DFB  6          ;DirName and VolName must be in the same page
BD1A: 06 59          ASC  "SETUPS"
BD1B: D3 C5 D4
BD1E: D5 D0 D3
60 ***** Get name of boot volume
BD21: AD 30 BF 61          ENTER  LDA  DEVNUM    ;get name of last volume accessed
BD24: 8D BF BE 62          STA  ONLINEN
BD27: 20 00 BF 63          JSR  PRODOS
BD2A: C5 64          DFB  $C5          ;ONLINE

```

```

BD2B: BE BE      65      DA      ONLINEP
BD2D: AD 01 BD    66      LDA      VOLNAME+1 ;insert a slash before the name
BD30: 29 0F      67      AND      #$0F
BD32: AA         68      TAX
BD33: E8         69      INX
BD34: 8E 00 BD    70      STX      VOLNAME
BD37: A9 2F      71      LDA      #$2F ;/
BD39: 8D 01 BD    72      STA      VOLNAME+1
BD3C: AD 04 BF    73      LDA      QUITVECT+1 ;save original quit vector
BD3F: 8D D2 BD    74      STA      QUITMOD1+1
BD42: AD 05 BF    75      LDA      QUITVECT+2
BD45: 8D D7 BD    76      STA      QUITMOD2+1
      77      ***** Clean up before & after calling files
BD48: A2 02      78      MAINLOOP LDX      #2 ;point Reset vector and ProDOS
BD4A: BD CB BD    79      LOOP3     LDA      JUMP+1,X ;Quit vectors to MainLoop
BD4D: 9D F2 03    80      STA      RESET,X
BD50: BD CA BD    81      LDA      JUMP,X
BD53: 9D 03 BF    82      STA      QUITVECT,X
BD56: CA         83      DEX
BD57: 10 F1      84      BPL      LOOP3
BD59: 9A         85      TXS          ;fix stack pointer (X=$FF)
BD5A: 20 85 BE    86      JSR      CLOSE ;close all open files
BD5D: A2 17      87      LDX      #23 ;clear system bit map
BD5F: A9 00      88      LDA      #0
BD61: 9D 58 BF    89      LOOP2     STA      BITMAP,X
BD64: CA         90      DEX
BD65: 10 FA      91      BPL      LOOP2
BD67: A9 CF      92      LDA      #$CF ;mark pages 0,1,4-7 as used
BD69: 8D 58 BF    93      STA      BITMAP
BD6C: A9 07      94      LDA      %111 ;mark pages $BD-$BF as used
BD6E: 8D 6F BF    95      STA      BITMAP+23
BD71: AD 82 C0    96      LDA      $C082 ;Language card off
BD74: 8D 0C C0    97      STA      $C00C ;40-column
BD77: 8D 0E C0    98      STA      $C00E ;normal character set
BD7A: 8D 00 C0    99      STA      $C000 ;80STORE off
BD7D: 20 84 FE    100     JSR      SETNORM ;normal
BD80: 20 2F FB    101     JSR      INIT ;display text page 1
BD83: 20 93 FE    102     JSR      SETVID ;PR#0
BD86: 20 89 FE    103     JSR      SETKBD ;IN#0
      104     *Make sure boot volume is around
      105     *AND set prefix to the boot volume
BD89: 20 58 FC    106     VOLMOUNT JSR      HOME
BD8C: 20 00 BF    107     JSR      PRODOS ;set prefix to volume
BD8F: C6         108     DFB      $C6 ;SET PREFIX
BD90: C5 BE      109     DA      PFX2P
BD92: 90 28      110     BCC      VOLOK
BD94: A2 0D      111     LDX      #13
BD96: BD F1 BE    112     LOOP6     LDA      VOLTEXT-1,X ;print message "insert volume"
BD99: 9D AC 05    113     STA      $5A8+4,X
BD9C: CA         114     DEX
BD9D: D0 F7      115     BNE      LOOP6
BD9F: BD 01 BD    116     LOOP7     LDA      VOLNAME+1,X ;print volume name
BDA2: 09 80      117     ORA      #$80
BDA4: 9D BB 05    118     STA      $5A8+19,X
BDA7: E8         119     INX
BDA8: EC 00 BD    120     CPX      VOLNAME
BDAB: 90 F2      121     BCC      LOOP7
BDAD: A9 23      122     LDA      #35 ;go to CH=35, CV=11
BDAF: 85 24      123     STA      CH
BDB1: A9 0B      124     LDA      #11
BDB3: 20 24 FC    125     JSR      VTABZ
BDB6: 20 0C FD    126     JSR      RDKEY ;wait for keypress
BDB9: 4C 89 BD    127     JMP      VOLMOUNT
      128     ***** Get name of next file at IN2
BDBC: 20 15 BE    129     VOLOK     JSR      NEXTFILE ;get name of next file at IN2
    
```

```

BDBF: B0 0D 130 BCS EXITLOOP ;if error, we're done with setup files
131 ***** Load and call setup file
BDC1: 20 00 BF 132 JSR PRODOS ;set prefix to SETUPS
BDC4: C6 133 DFB $C6 ;SET PREFIX
BDC5: C2 BE 134 DA PFX1P
BDC7: 20 8E BE 135 JSR READFILE ;read in file whose name is at IN@
136 ;and call it if there was no error.
BDCA: 4C 48 BD 137 JUMP JMP MAINLOOP ;3 bytes here copied into ProDOS quit vector
BDCD: 18 138 DFB #$BD!$A5 ;3 bytes here are copied into reset vector
BDCE: EE F4 03 139 EXITLOOP INC RESET+2 ;scramble reset vector
BDD1: A9 00 140 QUITMOD1 LDA #0 ;restore original quit vector
BDD3: 8D 04 BF 141 STA QUITVECT+1
BDD6: A9 00 142 QUITMOD2 LDA #0
BDD8: 8D 05 BF 143 STA QUITVECT+2
144 ***** Look for second system program on disk
BDD8: A9 00 145 LDA #0 ;modify NextFile routine so that it searches
BDDD: 8D 3E BE 146 STA NUMBER+1 ;the volume directory for system files only.
BDE0: 8D 82 BE 147 STA CHEKTYPE+1
BDE3: A9 00 148 LDA #<VOLNAME ;NamePtr+1 does not need to be changed
BDE5: 8D D2 BE 149 STA NAMEPTR ;since VolName and DirName are in the same
page
BDE8: 20 15 BE 150 NEXTSYS JSR NEXTFILE
BDEB: B0 1B 151 BCS QUIT
BDED: AE 80 02 152 LDX IN2 ;see if file ends with ".SYSTEM"
BDF0: A0 06 153 LDY #6
BDF2: BD 80 02 154 LOOP4 LDA IN2,X ;I expect pathname at IN2 in low ASCII
BDF5: D9 0E BE 155 CMP SYSTEXT,Y
BDF8: D0 EE 156 BNE NEXTSYS
BDFA: CA 157 DEX
BDFB: 88 158 DEY
BDFC: 10 F4 159 BPL LOOP4
BDFF: EE 02 BE 160 INC MOD+1
BE01: A9 FF 161 MOD LDA #$FF ;the first .SYSTEM program we find is this
BE03: F0 E3 162 BEQ NEXTSYS ;one, so skip it and look for next one.
BE05: 20 8E BE 163 JSR READFILE ;if successful, never come back
BE08: 20 00 BF 164 QUIT JSR PRODOS
BE0B: 65 165 DFB $65 ;QUIT
BE0C: C8 BE 166 DA QUITP
BE0E: 2E 53 59 167 SYSTEXT ASC '.SYSTEM'
BE11: 53 54 45 4D
168 * Get name of next system file or binary file
169 *
170 * This routine is set up to look for both SYSTEM and
171 * BINARY files in the SETUPS subdirectory. It is later
172 * modified to search for SYSTEM files only in the
173 * volume directory. The locations which are changed
174 * are ChekType+1, Number+1, and NamePtr (in the Open
175 * parametr list)
176 *
177 * Returns carry if not found, clear if found.
BE15: 20 00 BF 178 NEXTFILE JSR PRODOS
BE18: C8 179 DFB $C8 ;OPEN
BE19: D1 BE 180 DA OPENP
BE1B: B0 68 181 BCS CLOSE
BE1D: AD D6 BE 182 LDA OPENN
BE20: 8D D8 BE 183 STA MARKN
BE23: 8D DD BE 184 STA READN
BE26: 20 00 BF 185 JSR PRODOS ;Read in first 39 bytes of directory to
BE29: CA 186 DFB $CA ;IN2. This gets the number of entries per
BE2A: DC BE 187 DA READP ;block and number of bytes per entry.
BE2C: B0 57 188 BCS CLOSE
BE2E: AD A3 02 189 LDA IN2+35 ;save number of bytes per directory entry
BE31: 8D 54 BE 190 STA ENTSize+1
BE34: AD A4 02 191 LDA IN2+36 ;save number of entries per directory block
BE37: 8D 44 BE 192 STA ENTRIES+1

```



```

BE3A: EE 3E BE 193 NEXTENT INC NUMBER+1
BE3D: A9 00 194 NUMBER LDA #0 ;self-modified operand
195 * Retrieve catalog entry #A
BE3F: A2 FE 196 LDX #$FE ;build page index in X
BE41: E8 197 LOOP5 INX
BE42: E8 198 INX
BE43: C9 0D 199 ENTRIES CMP #13
BE45: 90 05 200 BCC OK
BE47: ED 44 BE 201 SBC ENTRIES+1
BE4A: B0 F5 202 BCS LOOP5 ;always
BE4C: A8 203 OK TAY
BE4D: A9 04 204 LDA #4 ;1st entry per directory block starts 4 bytes
in
BE4F: 88 205 LOOP10 DEY
BE50: 30 08 206 BMI OK2
BE52: 18 207 CLC
BE53: 69 27 208 ENTSIZE ADC #39 ;add size of directory entry
BE55: 90 F8 209 BCC LOOP10
BE57: E8 210 INX
BE58: D0 F5 211 BNE LOOP10 ;always
BE5A: 8D D9 BE 212 OK2 STA MARK ;save mark in file
BE5D: 8E DA BE 213 STX MARK+1
BE60: 20 00 BF 214 JSR PRODOS ;set the mark
BE63: CE 215 DFB $CE ;SET_MARK
BE64: D7 BE 216 DA MARKP
BE66: B0 1D 217 BCS CLOSE
BE68: 20 00 BF 218 JSR PRODOS ;read in directory info
BE6B: CA 219 DFB $CA ;READ
BE6C: DC BE 220 DA READP
BE6E: B0 15 221 BCS CLOSE
BE70: AD 80 02 222 LDA IN2 ;make sure that file is not deleted
BE73: F0 C5 223 BEQ NEXTENT
BE75: 29 0F 224 AND #$0F
BE77: 8D 80 02 225 STA IN2
BE7A: AD 90 02 226 LDA FILETYPE ;make sure file type is correct
BE7D: 49 FF 227 EOR $FF ;we look for system programs...
BE7F: F0 04 228 BEQ CLOSE
BE81: 49 F9 229 CHEKTYPE EOR #6!$FF ;...and binary ones.
BE83: D0 B5 230 BNE NEXTENT
BE85: 08 231 CLOSE PHP ;close all files - do not change carry
BE86: 20 00 BF 232 JSR PRODOS
BE89: CC 233 DFB $CC ;CLOSE
BE8A: CF BE 234 DA CLOSEP
BE8C: 28 235 PLP
BE8D: 60 236 ANRTS RTS
237 * Read file and call it.
238 * Name should be found at IN2
239 * Prefix must be set.
BE8E: AE 90 02 240 READFILE LDX FILETYPE ;if a system program, set to read to $2000
BE91: A9 20 241 LDA #$20
BE93: E8 242 INX
BE94: F0 06 243 BEQ SETDEST
BE96: AE 9F 02 244 LDX AUXCODE ;else, set to read in file at address
BE99: AD A0 02 245 LDA AUXCODE+1 ;found in auxcode
BE9C: 8E EC BE 246 SETDEST STX READ2D
BE9F: 8D ED BE 247 STA READ2D+1
BEA2: 20 00 BF 248 JSR PRODOS ;Open file
BEA5: C8 249 DFB $C8 ;OPEN
BEA6: E4 BE 250 DA OPEN2P
BEA8: B0 DB 251 BCS CLOSE
BEAA: AD E9 BE 252 LDA OPEN2N
BEAD: 8D EB BE 253 STA READ2N
BEB0: 20 00 BF 254 JSR PRODOS ;Read file into memory
BEB3: CA 255 DFB $CA ;READ
BEB4: EA BE 256 DA READ2P

```

```

DOS 3.3
257      JSR    CLOSE
258      BCS    ANRTS
259      JMP    (READ2D) ;call the file just loaded
260 ***** ProDOS MLI parameter lists
261 ONLINEP DFB 2 ;Online parameter list
262 ONLINEN DS 1
263 DA VOLNAME+1
264 *
BEC2: 01 265 PFX1P DFB 1 ;to set prefix to SETUP
BEC3: 1A BD 266 DA DIRNAME
267 *
BEC5: 01 268 PFX2P DFB 1 ;to set prefix to volume directory
BEC6: 00 BD 269 DA VOLNAME
270 *
BEC8: 04 00 00 271 QUITP DFB 4,0,0,0,0,0,0
BECB: 00 00 00 00
272 *
BECF: 01 00 273 CLOSEP DFB 1,0 ;close all files
274 *
BED1: 03 275 OPENP DFB 3 ;open directory
BED2: 1A BD 276 NAMEPTR DA DIRNAME ;pathname pointer
BED4: 00 B9 277 DA IOBUFFER
BED6: 00 278 OPENN DS 1 ;reference number
279 *
BED7: 02 280 MARKP DFB 2 ;set mark in directory
BED8: 00 281 MARKN DS 1
BED9: 00 00 00 282 MARK DS 3
283 *
BEDC: 04 284 READP DFB 4 ;read directory
BEDD: 00 285 READN DS 1
BEDE: 80 02 286 DA IN2 ;target address
BEE0: 27 00 287 DA 39 ;length
BEE2: 00 00 288 DS 2
289 *
BEE4: 03 290 OPEN2P DFB 3 ;open setup or system file
BEE5: 80 02 291 DA IN2
BEE7: 00 B9 292 DA IOBUFFER
BEE9: 00 293 OPEN2N DS 1
294 *
BEEA: 04 295 READ2P DFB 4 ;read setup or system file
BEEB: 00 296 READ2N DS 1
BEEC: 00 00 297 READ2D DS 2 ;destination of file is self-mod here
BEEE: 00 B1 298 DA $B900-$800 ;ask for largest possible that will fit
BEF0: 00 00 299 DS 2
300 *
BEF2: C9 CE D3 301 VOLTEXT ASC "INSERT VOLUME"
BEF5: C5 D2 D4 A0
BEF9: D6 CF CC D5
BEFD: CD C5
BEFF: E8 302 CHK ;checksum - eor for all previous bytes

--End assembly, 512 bytes, Errors: 0

```

No DOS Boot Sector

The following program can be ZAPped onto track 0, sector 0 of a DOS 3.3 disk to create a data disk. You can then change the disk free sector bitmap to use tracks 1 and 2 on your DOS 3.3 disks, giving you 32 extra sectors for data. See Chapter 4 of *Beneath Apple DOS* for information on the disk free sector bit map.

```

1      *****
2      *
3      * NO DOS BOOT
4      *
5      *****
6      *
7      ORG      $0800
8      SLOTNUM EQU    $2B
9      REBOOT  EQU    $FE
10     RESET   EQU    $03F4
11     MOTOROFF EQU    $C088
12     BELL    EQU    $FBDD
13     COUT    EQU    $FDED
14     GETKEY   EQU    $FD1B
15     HOME    EQU    $FC58
16     CV      EQU    $25
17     VTAB    EQU    $FC22
18     *
19     DFB      $01      ; LOAD ONLY 1 SECTOR FROM DISK (T0
0800: S0)
0801: A6 2B 20 ENTRY LDX  SLOTNUM      ; LEFT OVER FROM BOOT 0 ROM
0803: BD 88 C0 21 LDA  MOTOROFF,X ; TURN OFF THE MOTOR
0806: 8A 22 TXA
0807: 4A 23 LSR
0808: 4A 24 LSR
0809: 4A 25 LSR
080A: 4A 26 LSR
080B: 09 C0 27 ORA  #$C0      ; $0x -> $Cx
080D: 85 FF 28 STA  REBOOT+1 ; KEEP IT FOR LATER
080F: A9 00 29 LDA  #$00
0811: 85 FE 30 STA  REBOOT
0813: EE F4 03 31 INC  RESET      ; USER RESET => REBOOT
0816: 20 58 FC 32 JSR  HOME      ; CLEAR THE SCREEN
0819: 20 DD FB 33 JSR  BELL      ; BEEP TO ALERT USER
34      ; REMOVE PREVIOUS LINE FOR NO BELL
081C: A9 09 35 LDA  #$09      ; GO DOWN TO LINE 10
081E: 85 25 36 STA  CV          ; STORE IT THEN
0820: 20 22 FC 37 JSR  VTAB      ; SET BASL,H FOR PRINTING
0823: A2 00 38 LDX  #$00      ; START AT THE BEGINNING OF THE MSG
0825: BD 36 08 39 PRINT LDA  MSG,X ; GET EACH LETTER
0828: F0 06 40 BEQ  DONE      ; $00 MEANS WE'RE FINISHED
082A: 20 ED FD 41 JSR  COUT      ; PRINT IT
082D: E8 42 INX              ; ADVANCE TO NEXT CHARACTER

```

```

082E: D0 F5      43      BNE PRINT      ; LOOP TO PRINT IT
0830: 20 1B FD 44  DONE   JSR GETKEY     ; WAIT FOR KEYPRESS
0833: 6C FE 00 45      JMP (REBOOT) ; REBOOT SYSTEM
0836: A0 A0 A0 46  MSG    ASC "          THIS DISKETTE CANNOT BOOT."
0839: A0 A0 A0 A0 D4 C8 C9 D3
0841: A0 C4 C9 D3 CB C5 D4 D4
0849: C5 A0 C3 C1 CE CE CF D4
0851: A0 C2 CF CF D4 AE
0857: 8D          47      DFB $8D      ; CARRIAGE RETURN
0858: A0 A0 A0 48      ASC "          PLEASE PLACE A SYSTEM DISK"
085B: A0 A0 A0 A0 D0 CC C5 C1
0863: D3 C5 A0 D0 CC C1 C3 C5
086B: A0 C1 A0 D3 D9 D3 D4 C5
0873: CD A0 C4 C9 D3 CB
0879: 8D          49      DFB $8D      ; CARRIAGE RETURN
087A: A0 A0 A0 50      ASC "          IN DRIVE #1 AND PRESS <RETURN>"
087D: A0 A0 C9 CE A0 C4 D2 C9
0885: D6 C5 A0 A3 B1 A0 C1 CE
088D: C4 A0 D0 D2 C5 D3 D3 A0
0895: BC D2 C5 D4 D5 D2 CE BE
089D: 8D          51      DFB $8D      ; CARRIAGE RETURN
089E: 00          52      DFB $00      ; $00 MEANS END OF MESSAGE
089F: 00 00 00 53      DS $900-*
08A2: 00 00 00 00 00 00 00 00
08AA: 00 00 00 00 00 00 00 00
08B2: 00 00 00 00 00 00 00 00
08BA: 00 00 00 00 00 00 00 00
08C2: 00 00 00 00 00 00 00 00
08CA: 00 00 00 00 00 00 00 00
08D2: 00 00 00 00 00 00 00 00
08DA: 00 00 00 00 00 00 00 00
08E2: 00 00 00 00 00 00 00 00
08EA: 00 00 00 00 00 00 00 00
08F2: 00 00 00 00 00 00 00 00
08FA: 00 00 00 00 00 00 00 00

```

--End assembly, 256 bytes, Errors: 0

Sample Decryption Log

```
-----BurgerTime-----
A 4am crack                2015-12-31
-----
```

Name: BurgerTime
 Genre: arcade
 Year: 1982
 Author: Data East USA
 Publisher: Mattel
 Media: single-sided 5.25-inch floppy
 OS: DOS 3.3 with custom bootloader and
 custom file loader and custom RWTS,
 so not really like DOS 3.3 at all
 Previous cracks: The Freeze

~

Chapter 0
 In Which Various Automated Tools Fail
 In Interesting Ways

COPYA
 read error on first pass

Locksmith Fast Disk Backup
 bizarre combination of success and
 failure

--v--

LOCKSMITH 7.0 FAST DISK BACKUP

```
R.*****.***.***.*****
W
HEX 00000000000000001111111111111222
TRK 0123456789ABCDEF0123456789ABCDEF012
0.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
1.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
2.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
3.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
4.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
5.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
6.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
7.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
8.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
9.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
A.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
B.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
C.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
D.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
12 E.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
F.AAAAAAAAAA.AAA.....DDDDDDDDDDDDDD
[                               ] PRESS [RESET] TO EXIT
```

--^--

Tracks \$00, \$0B, \$0F, \$10, \$12, \$13,
 and \$14 are standard. Also a few

sectors of T15 and T22, and all of
 T11 except for what appears to be a
 bad sector in the middle. (The
 original disk boots fine, and the
 entire disk catalog on that track is
 fake anyway, so I think I got lucky.)

EDD 4 bit copy (no sync, no count)
 no errors, but copy displays an error
 "UNABLE TO LOAD GAME" and hangs

Copy][+ nibble editor
 T01-T0A, T0C-T0E are unformatted
 (hi-res disk scan confirms this)
 T15+ use a modified data prologue
 ("D5 AA AA" instead of "D5 AA AD")

Disk Fixer
 T00 -> custom bootloader
 T11 -> DOS 3.3 style disk catalog
 (fake, all files point to nothing)
 ["0" -> "Input/Output Control"]
 set Data Prologue to "D5 AA AA"
 Success! T15+ readable

Why didn't COPYA work?
 unformatted tracks, modified prologue

Why didn't Locksmith FDB work?
 modified prologue

Why didn't my EDD copy work?
 I assume there's some sort of
 protection check during boot. Disks
 don't say "UNABLE TO LOAD GAME"
 unless someone tells them to.

Next steps:

1. Trace the boot
2. ???

~

Chapter 1
 In Which We Start From Scratch

We're starting from bare metal on this
 one. My automated tools, they do nothing
 for us. Strap in.

[S6,D1=original disk]

[S6,D2=crack-in-progress (the partial
 copy I made with Locksmith Fast Disk
 Backup)]

[S5,D1=my work disk]

```

]PR#5
...
]CALL -151

*9600<C600.C6FFM

; copy boot sector (T00,S00) to the
; graphics page so it survives a reboot
96F8-  A0 00      LDY  #$00
96FA-  B9 00 08   LDA  $0800,Y
96FD-  99 00 20   STA  $2000,Y
9700-  C8         INY
9701-  D0 F7      BNE  $96FA

; turn off slot 6 drive motor
9703-  AD E8 C0   LDA  $C0E8

; reboot to my work disk in slot 5
9706-  4C 00 C5   JMP  $C500

*9600G
...reboots slot 6...
...reboots slot 5...

]BSAVE BOOT0,A,$2000
]CALL -151

*800<2000.20FFM

*801L

; starts off looking like a DOS 3.3
; bootloader -- zp$27 will be $09 the
; first time through, so this is a way
; to do one-time initialization
0801-  A5 27      LDA  $27
0803-  C9 09      CMP  #$09
0805-  D0 1B      BNE  $0822

; munge reset vector
0807-  A5 2B      LDA  $2B
0809-  8D F4 03   STA  $03F4

; munge boot slot into a vector to read
; more sectors
080C-  4A         LSR
080D-  4A         LSR
080E-  4A         LSR
080F-  4A         LSR
0810-  09 C0      ORA  #$C0
0812-  85 D7      STA  $D7
0814-  A9 5C      LDA  #$5C
0816-  85 D6      STA  $D6

Now ($D6) points to $Cx5C (based on the
boot slot, e.g. $C65C for slot 6). This
entry point will read a sector from
track $00 and exit via $0801.

As a bonus, setting $D6 to anything
other than 0 sets the RUN flag, so if I
managed to break to a BASIC prompt, any
command would run. Even trying to do a
"CALL -151" to get to the monitor would
be blocked. Lovely.

0818-  18         CLC
0819-  AD F9 08   LDA  $08F9
081C-  6D FA 08   ADC  $08FA
081F-  8D F9 08   STA  $08F9
0822-  AE FA 08   LDY  $08FA
0825-  F0 15      BEQ  $083C

```

```

*8F9.8FA

08F9-  03 04

We're loading 4 sectors directly into
the text page ($0400..$07FF). Nice.

; set up the rest of the zero page
; globals for the sector read routine
0827-  BD FB 08   LDA  $08FB,X
082A-  85 3D      STA  $3D
082C-  CE FA 08   DEC  $08FA
082F-  AD F9 08   LDA  $08F9
0832-  85 27      STA  $27
0834-  CE F9 08   DEC  $08F9

; read a sector (exits via $0801)
0837-  A6 2B      LDX  $2B
0839-  6C D6 00   JMP  ($00D6)

; Execution continues here (from $0825)
; after we've read all the sectors we
; wanted to read (into $0400..$07FF).
; Switch to the hi-res graphics screen
; (uninitialized).
083C-  2C 52 C0   BIT  $C052
083F-  2C 55 C0   BIT  $C055
0842-  2C 57 C0   BIT  $C057
0845-  2C 50 C0   BIT  $C050

; call part of the code we just read
0848-  20 0A 04   JSR  $040A

And that's where I need to interrupt
the boot.

```

Chapter 2 In Which We Eschew Obfuscation

```

*9600<C600.C6FFM

; set up callback after boot0 loads the
; code into the text page
96F8-  A9 4C      LDA  #$4C
96FA-  8D 48 08   STA  $0848
96FD-  A9 0A      LDA  #$0A
96FF-  8D 49 08   STA  $0849
9702-  A9 97      LDA  #$97
9704-  8D 4A 08   STA  $084A

; start the boot
9707-  4C 01 08   JMP  $0801

; callback is here --
; copy the text page to higher memory
; so it survives a reboot
970A-  A2 04      LDX  #$04
970C-  A0 00      LDY  #$00
970E-  B9 00 04   LDA  $0400,Y
9711-  99 00 24   STA  $2400,Y
9714-  C8         INY
9715-  D0 F7      BNE  $970E
9717-  EE 10 97   INC  $9710
971A-  EE 13 97   INC  $9713
971D-  CA         DEX
971E-  D0 EE      BNE  $970E

; turn off the slot 6 drive motor
9720-  AD E8 C0   LDA  $C0E8

```

```
; reboot to my work disk
9723- 4C 00 C5 JMP $C500
```

```
*BSAVE TRACE,A$9600,L$126
*9600G
```

```
...reboots slot 6...
...reboots slot 5...
```

```
]BSAVE BOOT1 0400-07FF,A$2400,L$400
]CALL -151
```

I'll need to leave this code at \$2400 for inspection, since I can't put it in the text page and also inspect it in the monitor. Relative branches will look correct, but absolute addresses will be off by \$2000.

```
*240AL
```

```
; wipe most of main memory, $0900+
```

```
240A- A0 09 LDY $09
240C- 84 01 STY $01
240E- A9 00 LDA $00
2410- 85 00 STA $00
2412- A8 TAY
2413- 91 00 STA ($00),Y
2415- C8 INY
2416- D0 FB BNE $2413
2418- E6 01 INC $01
241A- A6 01 LDX $01
241C- E0 C0 CPX $C0
241E- D0 F3 BNE $2413
2420- 60 RTS
```

Continuing in boot0, from \$084B...

```
*BLOAD BOOT0,A$800
```

```
*84BL
```

```
; set a (presumably unfriendly) reset
; vector
```

```
084B- A9 00 LDA $00
084D- 8D F2 03 STA $03F2
0850- A9 04 LDA $04
0852- 8D F3 03 STA $03F3
0855- 49 A5 EOR $A5
0857- 8D F4 03 STA $03F4
085A- EA NOP
085B- EA NOP
085C- EA NOP
085D- EA NOP
085E- 20 21 04 JSR $0421
```

```
*2421L
```

This code is exquisitely obfuscated, so let's just take it one line at a time. Everything is important.

```
2421- A9 00 LDA $00 ;A=00
2423- 85 82 STA $82 ;=00
```

; Stack pointer is uninitialized at this point (and not guaranteed to be anything in particular on boot), so I assume this is the first part of a verification that the stack pointer isn't being modified by anything the original developers didn't know about (like an NMI card).

```
2425- BA TSX
```

```
2426- 8A TXA
2427- 49 BC EOR $BC
2429- 85 50 STA $50

242B- 29 00 AND $00 ;A=00
242D- AA TAX ;X=00
242E- A0 D6 LDY $D6 ;Y=D6
2430- 49 54 EOR $54 ;A=54
2432- 84 81 STY $81 ;=D6
```

Now (\$81) points to \$00D6. That address was set to \$5C (at \$0816), because it was used as the pointer to \$Cx5C to read more sectors from disk (including the code we're now executing).

```
; X=00, so this is taking ($81), which
; is zp$D6, which is $5C, then EOR'ing
; it with A, which is $54
2434- 41 81 EOR ($81,X) ;A=08
2436- 85 21 STA $21 ;=08
2438- 98 TYA ;A=D6
2439- 49 E0 EOR $E0 ;A=36
243B- 85 81 STA $81 ;=36
```

Now (\$81) points to \$0036, the output vector.

```
243D- 49 A4 EOR $A4 ;A=92
243F- 81 81 STA ($81,X) ;=92
2441- E6 81 INC $81 ;=37
2443- 38 SEC
2444- E9 8B SBC $8B ;A=07
2446- 30 F0 BMI $2438 ;nope
2448- 81 81 STA ($81,X) ;=07
```

Now (\$36) points to \$0792.

```
244A- 49 07 EOR $07 ;A=00
244C- 85 20 STA $20 ;=00
```

Now (\$20) points to \$0800, which is the start of the boot0 code.

```
244E- 85 51 STA $51 ;=00
2450- 49 04 EOR $04 ;A=04
2452- F0 02 BEQ $2456 ;nope
2454- 85 46 STA $46 ;=04
2456- 8A TXA ;A=00
2457- 85 45 STA $45 ;=00
```

Now (\$45) points to \$0400, which is the start of the boot1 code.

```
2459- 18 CLC
245A- 69 D6 ADC $D6 ;A=D6
245C- 85 81 STA $81 ;=D6
```

Now (\$81) points to \$00D6 again.

```
245E- 4A LSR ;A=6B
245F- AA TAX ;X=6B
```

; \$16 + \$6B = \$81, and (\$81) points to \$00D6, and zp\$D6 is still \$5C.

```
2460- A1 16 LDA ($16,X) ;A=5C
2462- 49 32 EOR $32 ;A=6E
2464- A8 TAY ;Y=6E
2465- 49 6A EOR $6A ;A=04
2467- F0 E5 BEQ $244E ;nope
2469- 85 08 STA $08 ;=04
246B- 49 3E EOR $3E ;A=3A
246D- 88 DEY ;Y=6D
```

```
; ($45) points to $0400, and Y=$6D,
; so ($45),Y = $046D, which is $88 (the
; "DEY" instruction we just executed).
; A=$3A, and $3A EOR $88 = $B2.
246E- 51 45      EOR ($45),Y ;A=B2
```

```
; ($36) points to $0792, and Y=$6D,
; so ($36),Y = $07FF, which is $A0
; (trust me on that). A=$B2 after the
; previous EOR, so $B2 EOR $A0 = $12.
2470- 51 36      EOR ($36),Y ;A=12
2472- 38          SEC
```

```
; ($20) points to $0800, and Y=$6D,
; so ($20),Y = $086D, which is $09.
; A=$12 after the two previous EORs, so
; $12 - $09 = $09. (Think hex.)
2473- F1 20      SBC ($20),Y ;A=09
```

```
; store that in $07FF
2475- 91 36      STA ($36),Y ; =09
```

```
; done decrypting this page?
2477- C0 00      CPY #$00
```

```
; nope, branch back to finish it
2479- D0 F2      BNE $246D
```

That branch goes to \$046D, which is a "DEY" instruction, so we're decrypting memory from the top down.

```
247B- 84 81      STY $81 ; =00
```

```
; decrement page count (initialized to
; 04 at $0469)
247D- C6 08      DEC $08
```

```
; jump forward if we're done decrypting
247F- F0 05      BEQ $2486
```

```
; otherwise decrement the target page
; and loop back to decrypt it
2481- C6 37      DEC $37
2483- 4C 6D 04    JMP $046D
```

On the first pass, Y started at \$6D, so we only decrypted part of the page. But the index was offset from (\$36), which was \$0792 (set at \$0448), so we end up decrypting \$07FF..\$0792. Then zp\$37 is decremented and Y rolls over to \$FF, so we decrypt a full \$100 bytes in round 2 (\$0791..\$0692). Then \$0691..\$0592. Then \$0591..\$0492. Then zp\$08 hits 0 and we branch forward to \$0486.

```
; zp$50 was set from the uninitialized
; stack pointer (EOR'd with some magic
; number), and now we're using that as
; an index into the stack to get some
; other (also uninitialized) byte.
; Again, this will probably be checked
; later to see if the stack has been
; tampered with.
```

```
2486- A6 50      LDX $50
2488- BD 00 01    LDA $0100,X
248B- 85 46      STA $46
248D- 49 47      EOR #$47
248F- 8D 33 60    STA $6033
```

```
; everything after this was decrypted
```

```
; by the loop we just executed, so this
; listing is meaningless
2492- 64          ???
2493- 1E 11 B4    ASL $B411,X
2496- FE 36 96    INC $9636,X
2499- 70 FE      BVS $2499
```

To sum up: boot1 decrypts itself using both the calling code and itself as the decryption key, then immediately falls through to decrypted code (at \$0492). Patching boot0 to interrupt the boot will cause the decryption to fail. Patching boot1 to interrupt the boot will cause the decryption to fail.

I am stuck.

~

Chapter 3 In Which We Get Unstuck

The decryption routine at \$0421 is complicated and interconnected, but one thing it does not rely on is the caller address. I can simply call it myself and see what happens. As long as boot0 and boot1 are pristine by the time the routine is called, it should work.

*9600<C600.C6FFM

```
; set up callback after boot0 loads
; boot1
```

```
96F8- A9 4C      LDA #$4C
96FA- 8D 48 08    STA $0848
96FD- A9 0A      LDA #$0A
96FF- 8D 49 08    STA $0849
9702- A9 97      LDA #$97
9704- 8D 4A 08    STA $084A
```

```
; start the boot
9707- 4C 01 08    JMP $0801
```

```
; callback is here --
; undo the patch we made earlier, so
; boot0 is in a "pristine" state for
; the decryption routine
```

```
970A- A9 20      LDA #$20
970C- 8D 48 08    STA $0848
970F- A9 0A      LDA #$0A
9711- 8D 49 08    STA $0849
9714- A9 04      LDA #$04
9716- 8D 4A 08    STA $084A
```

```
; call the decryption routine
9719- 20 21 04    JSR $0421
```

```
; copy the (hopefully decrypted) boot1
; code to higher memory so it survives
; a reboot
```

```
971C- A2 04      LDX #$04
971E- A0 00      LDY #$00
9720- B9 00 04    LDA $0400,Y
9723- 99 00 24    STA $2400,Y
9726- C8          INY
9727- D0 F7      BNE $9720
9729- EE 22 97    INC $9722
972C- EE 25 97    INC $9725
972F- CA          DEX
9730- D0 EE      BNE $9720
```



```
; turn off slot 6 drive motor and
; reboot to my work disk
9732- AD E8 C0 LDA $C0E8
9735- 4C 00 C5 JMP $C500
```

```
*BSAVE TRACE2,A$9600,L$138
*9600G
...reboots slot 6...
...reboots slot 5...
```

```
]BSAVE BOOT1 DECRYPTED,A$2400,L$400
]CALL -151
```

Let's see what that decrypted code was that we were falling through to.

```
*2492L
```

```
2492- 60 RTS
```

Well OK then. I guess it worked.

~

Chapter 4 In Which We Have A Revelation

Continuing the trace from \$0861 (after calling the decryption routine)...

```
*BLOAD BOOT0,A$800
*861L
```

```
0861- A9 00 LDA #$00
0863- 85 24 STA $24
0865- 85 83 STA $83
0867- 85 00 STA $00
0869- 8D 78 02 STA $0278
086C- A9 09 LDA #$09
086E- 85 25 STA $25
0870- 85 01 STA $01
```

Not sure what all that's about, but (\$24) points to \$0900 and (\$00) points to \$0000.

```
; Hey, I have this decrypted now!
0872- 20 C0 04 JSR $04C0
```

```
*24C0L
```

```
; get boot slot (x16) -- generally a
; precursor to doing something disk-
; related
```

```
24C0- A6 2B LDX $2B
```

```
; (not shown) this subroutine divides
; X by 16 and puts it in Y, so if we
; booted from slot 6, Y is now 6
```

```
24C2- 20 6A 05 JSR $056A
```

```
; set up some stuff based on boot slot
; (this looks suspiciously like the
; kind of initialization that DOS 3.3
; does, tracking the current track of
; each disk in each slot+drive)
```

```
24C5- A9 00 LDA #$00
24C7- 99 78 02 STA $0278,Y
24CA- 99 88 02 STA $0288,Y
24CD- 8D 78 02 STA $0278
```

```
; don't know what these values mean,
; but they're probably important
```

```
24D0- A9 83 LDA #$83
24D2- 85 76 STA $76
24D4- A9 11 LDA $11
24D6- 85 80 STA $80
24D8- A9 07 LDA $07
24DA- 85 81 STA $81
24DC- 85 82 STA $82
24DE- 20 BE 06 JSR $06BE
```

```
*26BEL
```

```
26BE- 20 06 05 JSR $0506
```

```
*2506L
```

```
2506- A9 80 LDA #$80
2508- 85 02 STA $02
250A- A9 00 LDA $00
250C- 85 03 STA $03
```

Now (\$02) points to \$0080, which was set to \$11 at \$04D6.

```
250E- A0 00 LDY #$00 ;Y=00
2510- B1 02 LDA ($02),Y ;A=11
2512- 85 40 STA $40 ;=11
2514- C8 INY ;Y=01
2515- B1 02 LDA ($02),Y ;A=07
2517- 85 F2 STA $F2 ;=07
2519- C8 INY ;Y=02
251A- B1 02 LDA ($02),Y ;=07
251C- 85 3F STA $3F ;=07
251E- E6 02 INC $02
2520- E6 02 INC $02
2522- E6 02 INC $02
```

Now (\$02) points to \$0083, which was set to \$00 at \$0865.

```
2524- A9 D8 LDA #$D8 ;A=D8
2526- 85 47 STA $47 ;=D8
2528- A4 F2 LDY $F2 ;Y=07
```

```
; map of physical to logical sectors
```

```
252A- B9 5A 05 LDA $055A,Y
252D- 85 E9 STA $E9
252F- 20 71 05 JSR $0571
```

```
*2571L
```

```
2571- A6 2B LDX $2B
2573- 20 7A 05 JSR $057A
2576- 20 FF 05 JSR $05FF
2579- 60 RTS
```

I won't show \$057A, but it moves the drive arm to the track given in zp\$40, which was set to \$11 at \$0512.

```
*25FFL
```

```
; disk read routine that looks
; suspiciously like the one in the
; Disk II firmware (at $C65C)
```

```
25FF- 18 CLC
2600- 08 PHP
2601- BD 8C C0 LDA $C08C,X
2604- 10 FB BPL $2601
2606- 49 D5 EOR $D5
2608- D0 F7 BNE $2601
260A- BD 8C C0 LDA $C08C,X
```

```

260D- 10 FB      BPL  $260A
260F-  C9 AA      CMP  #$AA
2611-  D0 F3      BNE  $2606
2613-  EA         NOP
2614-  BD 8C C0    LDA  $C08C,X
2617-  10 FB      BPL  $2614
2619-  C9 96      CMP  #$96
261B-  F0 09      BEQ  $2626
261D-  28         PLP
261E-  90 DF      BCC  $25FF
2620-  49 AA      EOR  #$AA    <-- !
2622-  F0 25      BEQ  $2649
2624-  D0 D9      BNE  $25FF

```

```

. [rest of routine is uninteresting,
.  except to note that it stores the
.  sector data in ($24), which points
.  to $0900 (set at $086E)]
.

```

Two key takeaways here. First, we're reading the unreadable sector T11,S07. I originally thought this was a bad sector on my original disk, but I was wrong. Going back to it in a sector editor, it's perfectly readable if I set the data prologue to "D5 AA AA". It looks like this:

--v--

```

----- DISK EDIT -----
TRACK $11/SECTOR $07/VOLUME $FE/BYTE$00
$00:>FF<00 00 11 07 00 00 00 .@@QG@@@
$08: 00 00 00 22 0F 02 C8 C5 @@@OBHE
$10: CC CC CF A0 A0 A0 A0 A0 LLO
$18: A0 A0 A0 A0 A0 A0 A0 A0
$20: A0 A0 A0 A0 A0 A0 A0 A0
$28: A0 A0 A0 A0 02 00 22 0D B@"M
$30: 04 C2 D5 D2 C7 C5 D2 D4 DBURGERT
$38: C9 CD C5 A0 A0 A0 A0 A0 IME
$40: A0 A0 A0 A0 A0 A0 A0 A0
$48: A0 A0 A0 A0 A0 A0 A0 B5
$50: 00 17 08 04 CD C4 D3 C1 @WHDMSA
$58: C4 CA A0 A0 A0 A0 A0 A0 DJ
$60: A0 A0 A0 A0 A0 A0 A0 A0
$68: A0 A0 A0 A0 A0 A0 A0 A0
$70: A0 A0 26 00 00 00 00 00 &@@@@@
$78: 00 00 00 00 00 00 00 00 @@@@@@@@

```

BUFFER 0/SLOT 6/DRIVE 1/MASK OFF/NORMAL

COMMAND : _

--^--

That looks *exactly* like a DOS 3.3 catalog sector. I'll come back to that revelation later.

Takeaway #2: the RWTS that reads this sector (and presumably the rest of the disk) is tucked away behind the insane decryption routine at \$0421. In order to modify it to read a standard format disk (by changing \$0621 from \$AA to \$AD to match the third nibble of the data prologue), I'll need to write the unencrypted bootloader back to disk and disable the decryption routine.

~

Chapter 5 In Which We Make The Sort Of Progress That Doesn't Feel Like Progress

]PR#5

...
]CALL -151

```

; straightforward multi-sector write
; loop, via the RWTS vector at $03D9
08C0- A9 08      LDA  #$08
08C2- A0 E8      LDY  #$E8
08C4- 20 D9 03    JSR  $03D9
08C7- AC ED 08    LDY  $08ED
08CA- 88         DEY
08CB- 10 05      BPL  $08D2
08CD- A0 0F      LDY  #$0F
08CF- CE EC 08    DEC  $08EC
08D2- 8C ED 08    STY  $08ED
08D5- CE F1 08    DEC  $08F1
08D8- CE E1 08    DEC  $08E1
08DB- D0 E3      BNE  $08C0
08DD- 60         RTS

```

*8E0.8FF

08E0- 00 05 00 00 00 00 00 00

sector count

08E8- 01 60 01 00 00 04 FB 08
 ^^ ^^ ^^^
 S6 D1 T0 S4

08F0- 00 27 00 00 02 00 FE 60
 ^^^^ ^^^

address write

08F8- 01 00 00 00 01 EF D8 00

*BSAVE WRITE BOOT1,A\$8C0,L\$40

; load decrypted bootloader

*BLOAD BOOT0,A\$2300

*BLOAD BOOT1 DECRYPTED,A\$2400

```

; disable decryption loop (we'll still
; let it run, but this disables the
; actual "STA" instruction at $0475,
; so it won't overwrite the already-
; decrypted code we're about to write
; to disk)
*2475:24

```

```

; normalize RWTS by fixing the third
; nibble of the data prologue
*2621:AD

```

[S6,D1=crack-in-progress (the partial copy I made with Locksmith Fast Disk Backup)]

```

; write decrypted+patched bootloader
; to disk
*8C0G
...write write write...

```

Now I need to convert the protected tracks that use the non-standard data prologue. Super Demuffin only converts

whole tracks, but track \$15 and \$22 are a mix of protected and unprotected sectors. I'll need to make an RWTS file and pump it through Advanced Demuffin.

[S6,D1=DOS 3.3 system master]

]PR#6

...

]CALL -151

; copy RWTS

*3800<B800.BFFFM

; modify third nibble of data prologue

*38FC:AA

*BSAVE RWTS,A\$3800,L\$800,S5,D1

*BRUN ADVANCED DEMUFFIN 1.5,S5,D1

[S6,D1=original disk]

[S6,D2=crack-in-progress]

[press "5" to switch to slot 5]

[press "R" to load a new RWTS module]

--> At \$B8, load "RWTS" from drive 1

[press "6" to switch to slot 6]

[press "C" to convert disk]

[press "Y" to change default values]

--v--

ADVANCED DEMUFFIN 1.5 (C) 1983, 2014
ORIGINAL BY THE STACK UPDATES BY 4AM

=====

INPUT ALL VALUES IN HEX

SECTORS PER TRACK? (13/16) 16

START TRACK: \$15 <-- change this

START SECTOR: \$03 <-- change this

END TRACK: \$22

END SECTOR: \$0D <-- change this

INCREMENT: 1

MAX # OF RETRIES: 0

COPY FROM DRIVE 1

TO DRIVE: 2

=====

16SC \$15,\$03-\$22,\$0D BY\$01 S6,D1->S6,D2

--^--

And here we go...

--v--

ADVANCED DEMUFFIN 1.5 (C) 1983, 2014
ORIGINAL BY THE STACK UPDATES BY 4AM

=====PRESS ANY KEY TO CONTINUE=====

TRK:

+5:

0123456789ABCDEF0123456789ABCDEF012

SC0:

SC1:

SC2:

SC3:

SC4:

SC5:

SC6:

SC7:

SC8:

SC9:

SCA:

SCB:

SCC:

SCD:

SCE:

SCF:

=====

16SC \$15,\$03-\$22,\$0D BY\$01 S6,D1->S6,D2

--^--

And another conversion with the same
RWTS, this time for that one protected
sector on track \$11.

--v--

ADVANCED DEMUFFIN 1.5 (C) 1983, 2014
ORIGINAL BY THE STACK UPDATES BY 4AM

=====PRESS ANY KEY TO CONTINUE=====

TRK:

+5:

0123456789ABCDEF0123456789ABCDEF012

SC0:

SC1:

SC2:

SC3:

SC4:

SC5:

SC6:

SC7:

SC8:

SC9:

SCA:

SCB:

SCC:

SCD:

SCE:

SCF:

=====

16SC \$11,\$07-\$11,\$07 BY\$01 S6,D1->S6,D2

--^--

[S6,D1=crack-in-progress]

]PR#6

..."UNABLE TO LOAD GAME"...

There is an explicit protection check
somewhere after it reads T11,\$07. Now
that I've decrypted the bootloader and
patched the RWTS, it's finally able to
get far enough to detect that I'm
running an unauthorized copy.

In other words, I now have a COPYA-able
copy that is just as broken as my EDD
bit copy. This we call progress.

~

Chapter 6
In Which We Find That Which
Does Not Wish To Be Found

Picking up where we left off, at \$06C1:

```

]PR#5
...
]BLOAD BOOT1 DECRYPTED,A$2400
]CALL -151

*26C1L

26C1-  A6 2B      LDX    $2B

; I'll return to this in a moment
26C3-  20 FA 06    JSR    $06FA

; roll one bit of the accumulator into
; the carry
26C6-  2A          ROL

; if carry is clear, skip over the
; following code
26C7-  90 2F      BCC    $26F8

; turn off the drive motor
26C9-  A4 2B      LDY    $2B
26CB-  B9 88 C0    LDA    $C088,Y

; clear text page 2
26CE-  A9 A0      LDA    #$A0
26D0-  A2 00      LDX    #$00
26D2-  9D 00 08    STA    $0800,X
26D5-  E8          INX
26D6-  D0 FA      BNE    $26D2
26D8-  EE D4 06    INC    $06D4
26DB-  AC D4 06    LDY    $06D4
26DE-  C0 0C      CPY    #$0C
26E0-  D0 F0      BNE    $26D2

; display a message
26E2-  A0 00      LDY    #$00
26E4-  B9 89 07    LDA    $0789,Y
26E7-  99 00 08    STA    $0800,Y
26EA-  C8          INY
26EB-  C0 28      CPY    #$28
26ED-  D0 F5      BNE    $26E4

; show it
26EF-  AD 51 C0    LDA    $C051
26F2-  AD 55 C0    LDA    $C055

; jump to The Badlands
26F5-  4C DF 07    JMP    $07DF

; execution continues here (from $06C7)
26F8-  60          RTS

Here's the message that is displayed
at $06E2:

*FC58G N 400<2789.27B0M

UNABLE TO LOAD GAME

Don't look now, but I think we just
found the copy protection.

*26FAL

26FA-  BD 89 C0    LDA    $C089,X

```

```

26FD-  A9 56      LDA    #$56
26FF-  85 11      STA    $11
2701-  D0 01      BNE    $2704
2703-  D0 C6      BNE    $26CB
2705-  12          ???

```

Oh joy, more obfuscated code. The "BNE" at \$0701 unconditionally branches into the middle of the next instruction, which confuses the monitor's built-in disassembler.

```

*2703:EA
*26FAL

```

```

26FA-  BD 89 C0    LDA    $C089,X
26FD-  A9 56      LDA    #$56
26FF-  85 11      STA    $11
2701-  D0 01      BNE    $2704
2703-  EA          NOP
2704-  C6 12      DEC    $12
2706-  F0 03      BEQ    $270B
2708-  D0 0A      BNE    $2714
270A-  D0 C6      BNE    $26D2
270C-  11 D0      ORA    ($D0),Y

```

Oops, there's another one. \$0706 is a branch to \$070B, which is shown in the middle of an instruction again.

```
*270A:EA
```

I'll spare you the gory details, but there are a few more of these. Just a few. 16. There are 16 more.

```

*2713:EA
*271B:EA
*2722:EA
*272A:EA
*2731:EA
*2739:EA
*2740:EA
*274A:EA
*2751:EA
*2754:EA
*275E:EA
*2766:EA
*276E:EA
*2776:EA
*277D:EA
*2786:EA

```

Here's the final, as-unobfuscated-as-I-can-make-it copy protection routine:

```
*26FAL
```

```

; turn on drive motor
26FA-  BD 89 C0    LDA    $C089,X

; set up Death Counters
26FD-  A9 56      LDA    #$56
26FF-  85 11      STA    $11
2701-  D0 01      BNE    $2704
2703-  EA          NOP

```

```

; this is actually uninitialized, but
; it doesn't much matter because it's
; just the low byte of the 16-bit Death
; Counter
2704-  C6 12      DEC    $12
2706-  F0 03      BEQ    $270B

```

```

2708- D0 0A      BNE    $2714
270A- EA        NOP
270B- C6 11      DEC     $11
270D- D0 05      BNE    $2714

; if Death Counter hits 0, set A=01 and
; exit
270F- A9 01      LDA     #$01
2711- D0 74      BNE    $2787
2713- EA        NOP

```

```

; find standard address prologue
; (D5 AA 96)
2714- BD 8C C0    LDA     $C08C,X
2717- 10 FB      BPL     $2714
2719- D0 01      BNE    $271C
271B- EA        NOP
271C- C9 D5      CMP     #$D5
271E- F0 03      BEQ     $2723
2720- D0 E2      BNE    $2704
2722- EA        NOP
2723- BD 8C C0    LDA     $C08C,X
2726- 10 FB      BPL     $2723
2728- D0 01      BNE    $272B
272A- EA        NOP
272B- C9 AA      CMP     #$AA
272D- F0 03      BEQ     $2732
272F- D0 D3      BNE    $2704
2731- EA        NOP
2732- BD 8C C0    LDA     $C08C,X
2735- 10 FB      BPL     $2732
2737- D0 01      BNE    $273A
2739- EA        NOP
273A- C9 96      CMP     #$96
273C- F0 03      BEQ     $2741
273E- D0 C4      BNE    $2704
2740- EA        NOP

```

```

; skip over an $FF nibble
2741- A0 0A      LDY     #$0A
2743- BD 8C C0    LDA     $C08C,X
2746- 10 FB      BPL     $2743
2748- D0 01      BNE    $274B
274A- EA        NOP
274B- C9 FF      CMP     #$FF
274D- D0 03      BNE    $2752
274F- F0 B3      BEQ     $2704
2751- EA        NOP
2752- F0 01      BEQ     $2755
2754- EA        NOP

```

```

; Read data latch exactly once (no BPL
; loop here!) and make sure its value
; is correct. We're out of sync here
; because of all the branching, so the
; exact value of the data latch depends
; on timing bit after the $FF nibble.
; This is where my EDD bit copy failed.
2755- BD 8C C0    LDA     $C08C,X
2758- C9 A5      CMP     #$A5
275A- D0 A8      BNE    $2704
275C- F0 01      BEQ     $275F
275E- EA        NOP

```

```

; calculate a checksum on the following
; nibbles
275F- BD 8C C0    LDA     $C08C,X
2762- 10 FB      BPL     $275F
2764- D0 01      BNE    $2767
2766- EA        NOP
2767- 85 10      STA     $10
2769- 88        DEY
276A- D0 03      BNE    $276F

```

```

276C- F0 10      BEQ     $277E
276E- EA        NOP
276F- BD 8C C0    LDA     $C08C,X
2772- 10 FB      BPL     $276F
2774- D0 01      BNE    $2777
2776- EA        NOP
2777- 45 10      EOR     $10
2779- D0 EC      BNE    $2767
277B- F0 EA      BEQ     $2767
277D- EA        NOP

```

```

; final checksum must be $60
277E- A5 10      LDA     $10
2780- 49 60      EOR     #$60
2782- D0 80      BNE    $2704
2784- F0 01      BEQ     $2787
2786- EA        NOP

```

```

; on exit, A=00 on success (after
; falling through) or 01 on failure
; (coming from $0711)
2787- 60        RTS

```

This routine always returns to the caller, and the accumulator indicates success (0) or failure (1).

At \$06C6, I can change the "ROL" to a "CLC". Then the "BCC" at \$06C7 will unconditionally branch to \$06F8, as if the protection routine had passed.

T00,S03,\$C6 change 2A to 18

]PR#6
...offers input selection, then hangs with the drive motor on...

That's actually a lot of progress. I can now calibrate my joystick before the game tells me to go f--- myself.

~

Chapter 7 In Which I'd Like To Add You To My Professional Network Of Linked Catalog Sectors

Let's go back to that "unreadable" sector on track \$11, the one that looks exactly like a DOS 3.3 catalog sector, because it is a DOS 3.3 catalog sector. It's just not linked into the VTOC.

T11,S00 currently points to T11,S0F as the first catalog sector, but all of the "files" in that catalog sector are fake. What if I changed it to point to sector \$07 instead?

T11,S00,\$02 change 0F to 07

]PR#5
...
]CATALOG,S6,D1

C1983 DSR^C#254
324 FREE

A 002 HELLO
B 181 BURGERTIME

B 038 MDSADJ

Well would you look at that.

```
]BRUN BURGERTIME
...crashes...
```

It was worth a shot.

```
]PR#5
...
]BLOAD BURGERTIME,S6,D1
```

This actually works, but only because my work disk is running Diversi-DOS 64K which relocates most of DOS to the language card. The file starts at \$0C00 and is \$B200 in length. Whatever is at \$0C00 isn't code, which explains why I couldn't just BRUN it. But this is definitely (at least part of) the game code, because switching to the hi-res graphics screen shows the game's title screen.

```
]BRUN MDSADJ,S6,D1
...displays input selection screen,
  allows me to actually calibrate my
  joystick, then hangs with the drive
  motor on...
```

These files are definitely not fake. The bootloader is actually using this (previously hidden, but well-formed) catalog sector to load the input selection routine (MDSADJ), then doing it again to load the actual game. But something is getting stuck between those two, so I need to look at MDSADJ to find out what's going on.

```
]BLOAD MDSADJ
]CALL -151
```

*BF55.BF56

BF55- 00 64 ; start address

*6400L

```
6400- A9 9D      LDA    #$9D      ;A=9D
6402- 85 50      STA    $50       ; =9D
6404- D0 01      BNE    $6407     ;yes
6406- 75 A9      ADC     $A9,X
6408- 86 A2      STX     $A2
640A- 00         BRK
```

Right out of the gate, I can tell this is going to be fun(*). The branch at \$6404 jumps into the middle of the next instruction, which confuses the monitor disassembler.

(*) not guaranteed, actual fun may vary

*6407L

```
6407- A9 86      LDA    #$86      ;A=86
6409- A2 00      LDX     #$00      ;X=00
640B- F0 01      BEQ     $640E     ;yes
640D- 24 95      BIT     $95
```

And again.

*640EL

```
640E- 95 21      STA    $21,X
6410- 86 20      STX     $20
```

Now (\$20) points to \$8600.

```
6412- A9 01      LDA    #$01      ;A=01
6414- D0 01      BNE    $6417     ;yes
6416- D0 6C      BNE    $6484
```

And again.

*6417L

```
6417- 6C 20 00   JMP     ($0020) ;8600
```

*8600L

; decrypt everything in the file we
; just loaded (\$6400..\$85FF)

```
8600- A2 22      LDX     #$22
8602- A0 00      LDY     #$00
8604- B9 00 85   LDA     $8500,Y
8607- 59 00 84   EOR     $8400,Y
860A- 49 01      EOR     #$01
860C- 99 00 85   STA     $8500,Y
860F- C8        INY
8610- D0 F2      BNE     $8604
8612- CE 06 86   DEC     $8606
8615- CE 09 86   DEC     $8609
8618- CE 0E 86   DEC     $860E
861B- CA        DEX
861C- D0 E6      BNE     $8604
```

; and continue with decrypted code
861E- 4C 1D 64 JMP \$641D

Luckily (for me), this decryption loop is self-contained and does not rely on itself as a decryption key. I should be able to put an "RTS" at \$861E and run it from the monitor.

*861E:60

*8600G

*641DL

```
641D- 8D 5A DB   STA     $DB5A
6420- 49 70      EOR     #$70
6422- E8        INX
6423- 43        ???
6424- 4F        ???
6425- C9 28      CMP     #$28
6427- BF        ???
```

I've missed something.

Wait. On the last pass, it's decrypting \$6400..\$64FF by EOR'ing it with the page under that, which is \$6300..\$63FF. Which is not part of this file. Which means that this decryption loop is not self-contained after all; it depends on the value of the page at \$6300.

Backtracking the boot, the only thing that ever touched \$6300 was the routine at \$040A that wiped all of main memory with zeroes.

*BLOAD MDSADJ

```
*6300:00 N 6301<6300.63FEM
```

```
*861E:60
```

```
*8600G
```

```
*641DL
```

```
641D- 20 5D 64 JSR $645D
6420- 20 00 65 JSR $6500
6423- 20 2C 64 JSR $642C
6426- 20 00 85 JSR $8500
6429- 4C B0 68 JMP $68B0
642C- A6 2B LDX $2B
642E- BD 8E C0 LDA $C08E,X
6431- BD 8C C0 LDA $C08C,X
6434- BD 89 C0 LDA $C089,X
```

```
Bingo.
```

```
*BSAVE MDSADJ DECRYPTED,A$6400,L$2400,
S5,D1
```

Chapter 8
In Which We Discover
A Decryption Most Foul
(Again)

```
Continuing from $641D...
```

```
*641DL
```

```
641D- 20 5D 64 JSR $645D
```

```
*645DL
```

```
; wipe language card
```

```
645D- AD 83 C0 LDA $C083
6460- AD 83 C0 LDA $C083
6463- A0 00 LDY #$00
6465- 84 00 STY $00
6467- A9 D0 LDA #$D0
6469- 85 01 STA $01
646B- 91 00 STA ($00),Y
646D- C8 INY
646E- D0 FB BNE $646B
6470- E6 01 INC $01
6472- D0 F7 BNE $646B
```

```
; back to ROM
```

```
6474- AD 81 C0 LDA $C081
```

```
; and over here, then out
```

```
6477- 20 80 64 JSR $6480
647A- 60 RTS
```

```
*6480L
```

```
; Scan all peripheral (slot) ROMs, test
; whether the first byte of the slot
; ROM changes over a short period of
; time, and if not, save it to an array
; at $BF73. (I'm assuming this is done
; again later, and if the values don't
; match, the game knows that it's not
; running on the same machine -- i.e.
; that someone used a memory capture
; card to save the game code to disk
; then reload it without going through
; this bootloader.)
```

```
6480- A2 07 LDX #$07
6482- A9 00 LDA #$00
6484- 8D 9B 64 STA $649B
```

```
6487- 8D 9E 64 STA $649E
648A- 8A TXA
648B- 18 CLC
648C- 69 C0 ADC #$C0
648E- 8D 9C 64 STA $649C
6491- 8D 9F 64 STA $649F
6494- A9 0A LDA #$0A
6496- 85 A3 STA $A3
6498- A0 00 LDY #$00
649A- AD 00 C0 LDA $C000
649D- CD 00 C0 CMP $C000
64A0- D0 0E BNE $64B0
64A2- 88 DEY
64A3- D0 F8 BNE $649D
64A5- C6 A3 DEC $A3
64A7- D0 EF BNE $6498
64A9- 9D 73 BF STA $BF73,X
64AC- CA DEX
64AD- D0 D3 BNE $6482
64AF- 60 RTS
64B0- A9 00 LDA #$00
64B2- 4C A9 64 JMP $64A9
```

Since I'm planning to retain this entire bootloader (albeit without the decrypt-y bits or the protect-y bits), I won't bother to disable it now. But if it turns out I can't retain the bootloader, I'll need to revisit it.

```
Continuing from $6420...
```

```
*6420L
```

```
; (not shown) This is the actual input
; selection routine that sets joystick
; or keyboard mode. It returns when
; the user has made a selection or time
; has run out.
```

```
6420- 20 00 65 JSR $6500
```

Once that returns, things suddenly get a lot more interesting (for me, at least).

```
6423- 20 2C 64 JSR $642C
```

```
*642CL
```

```
; turn on drive motor
```

```
642C- A6 2B LDX $2B
642E- BD 8E C0 LDA $C08E,X
6431- BD 8C C0 LDA $C08C,X
6434- BD 89 C0 LDA $C089,X
```

```
; clear hi-res screen 2
```

```
6437- A9 00 LDA #$00
6439- 85 00 STA $00
643B- A0 40 LDY $40
643D- 84 01 STY $01
643F- A8 TAY
6440- 91 00 STA ($00),Y
6442- C8 INY
6443- D0 FB BNE $6440
6445- E6 01 INC $01
6447- A6 01 LDX $01
6449- E0 60 CPX $60
644B- D0 F3 BNE $6440
```

```
; and show it (blank)
```

```
644D- AD 50 C0 LDA $C050
6450- AD 52 C0 LDA $C052
6453- AD 55 C0 LDA $C055
```

```

6456- AD 57 C0 LDA $C057
; clear keyboard strobe
6459- AD 10 C0 LDA $C010
645C- 60 RTS
Continuing from $6426...
*6426L
6426- 20 00 85 JSR $8500
*8500L
; oh God, here we go again
8500- A9 00 LDA #$00 ;A=00
8502- 85 82 STA $82 ;=00
; save stack pointer again
8504- BA TSX
8505- 8A TXA
8506- 49 BC EOR #$BC
8508- 85 50 STA $50
850A- 29 00 AND #$00 ;A=00
850C- AA TAX ;X=00
850D- AD D6 LDY #$D6 ;Y=D6
850F- 49 3A EOR #$3A ;A=3A
8511- 84 81 STY $81 ;=D6
Now ($81) points to $00D6. That address
was set to $5C (at $0816) and has never
been touched since.
; X=00, so this is taking ($81), which
; is zp$D6, which is $5C, then EOR'ing
; it with A, which is $3A
8513- 41 81 EOR ($81,X) ;A=66
8515- 85 01 STA $01 ;=66
Now ($00) points to $6600.
8517- 98 TYA ;A=D6
8518- 49 D2 EOR #$D2 ;A=04
851A- 85 81 STA $81 ;=04
Now ($81) points to $0004.
851C- 49 04 EOR #$04 ;A=00
; ($81) points to $0004 and X=00, so
; this sets zp$04 to $00.
851E- 81 81 STA ($81,X) ;=00
8520- E6 81 INC $81
Now ($81) points to $0005.
8522- 38 SEC
8523- E9 7D SBC #$7D ;A=83
8525- 10 F0 BPL $8517 ;nope
; ($81) points to $0005 and X=00, so
; this sets zp$05 to $83, so now ($04)
; points to $8300.
8527- 81 81 STA ($81,X) ;=83
8529- 49 83 EOR #$83 ;A=00
852B- 85 00 STA $00 ;=00
852D- 85 51 STA $51 ;=00
852F- 49 85 EOR #$85 ;A=85
8531- F0 02 BEQ $8535 ;nope
8533- 85 03 STA $03 ;=85
8535- 8A TXA ;A=00
8536- 85 02 STA $02 ;=00

```

Now (\$02) points to \$8500.

```

8538- 18 CLC
8539- 69 D6 ADC #$D6 ;A=D6
853B- 85 81 STA $81 ;=86

```

Now (\$81) points to \$00D6 again.

```

853D- 4A LSR ;A=6B
853E- AA TAX ;X=6B

```

; \$16 + \$6B = \$81, and (\$81) points to
; \$00D6, and zp\$D6 is still \$5C.

```

853F- A1 16 LDA ($16,X) ;A=5C
8541- 49 5C EOR #$5C ;A=00
8543- A8 TAY ;Y=00
8544- 49 04 EOR #$04 ;A=04
8546- F0 E5 BEQ $852D ;nope
8548- 85 08 STA $08 ;=04
854A- 49 7A EOR #$7A ;A=7E
854C- 88 DEY ;Y=FF

```

; EOR with (\$02), Y = \$8500, Y = \$85FF
854D- 51 02 EOR (\$02), Y

; EOR with (\$04), Y = \$8300, Y = \$83FF
854F- 51 04 EOR (\$04), Y

; SBC (\$00), Y = \$6600, Y = \$66FF
8551- 38 SEC
8552- F1 00 SBC (\$00), Y

; store it in (\$04), Y = \$8300, Y = \$83FF
8554- 91 04 STA (\$04), Y

; done decrypting this page?
8556- C0 00 CPY #\$00

; nope, branch back to finish it
8558- D0 F2 BNE \$854C

That branch goes to \$854C, which is a
"DEY" instruction, so we're decrypting
memory from the top down. This is the
same basic pattern as the earlier
decryption routine at \$0421, which was
equally insane.

```

855A- 84 81 STY $81 ;=00

```

; decrement page count (initialized to
; 04 at \$8548)
855C- C6 08 DEC \$08

; jump forward if we're done decrypting
855E- F0 05 BEQ \$8565

; otherwise decrement the target page
; and loop back to decrypt it
8560- C6 05 DEC \$05
8562- 4C 4C 85 JMP \$854C

The first pass decrypts \$83FF..\$8300
against \$85FF..\$8500 and \$66FF..\$6600.
The second pass decrypts \$82FF..\$8200;
then \$81FF..\$8100; then \$80FF..\$8000.
Then zp\$08 hits 0 and we branch forward
to \$8565.

```

; get some byte from the stack
8565- A6 50 LDX $50
8567- BD 00 01 LDA $0100,X

```



```

856A- 85 03      STA  $03
856C- 49 47      EOR  #$47
856E- 60         RTS

```

This routine is almost self-contained. The only thing it relies on to be pre-initialized is zero page \$D6, which must be \$5C. Other than that, I can run it directly from the monitor.

*D6:5C N 8500G

*8000L

```

8000- A1 A9      LDA  ($A9,X)
8002- 09 85      ORA  #$85
8004- 25 85      AND  $85
8006- 01 A9      ORA  ($A9,X)
8008- 00         BRK
8009- 85 24      STA  $24
800B- 85 83      STA  $83
800D- 85 00      STA  $00
800F- 20 C0 04   JSR  $04C0

```

That *almost* looks like real code. Wait, I see it now.

*8001L

```

8001- A9 09      LDA  #$09
8003- 85 25      STA  $25
8005- 85 01      STA  $01
8007- A9 00      LDA  #$00
8009- 85 24      STA  $24
800B- 85 83      STA  $83
800D- 85 00      STA  $00
800F- 20 C0 04   JSR  $04C0

```

There we go.

*BSAVE MDSADJ DECRYPTED TWICE,A\$6400,
L\$2400,S5,D1

~

Chapter 9

In Which We'll Think About It After,
Like An After-Thought

Continuing from... um... \$6429...

*6429L

```

6429- 4C B0 68   JMP  $68B0

```

*68B0L

; copy newly decrypted code to the text
; page (overwriting the original RWTS)

```

68B0- A9 00      LDA  #$00
68B2- 85 00      STA  $00
68B4- A9 04      LDA  #$04
68B6- 85 01      STA  $01
68B8- A9 00      LDA  #$00
68BA- 85 02      STA  $02
68BC- A9 80      LDA  #$80
68BE- 85 03      STA  $03
68C0- A2 05      LDX  #$05
68C2- A0 00      LDY  #$00
68C4- B1 02      LDA  ($02),Y
68C6- 91 00      STA  ($00),Y
68C8- C8         INY

```

```

68C9- D0 F9      BNE  $68C4
68CB- CA         DEX
68CC- F0 07      BEQ  $68D5
68CE- E6 03      INC  $03
68D0- E6 01      INC  $01
68D2- 4C C4 68   JMP  $68C4

```

; execution continue here (from \$68CC)
; and we set yet another reset vector

```

68D5- A9 96      LDA  #$96
68D7- 8D F2 03   STA  $03F2
68DA- A9 04      LDA  #$04
68DC- 8D F3 03   STA  $03F3
68DF- 49 A5      EOR  $A5
68E1- 8D F4 03   STA  $03F4

```

; destroy the code we just decrypted
; and copied (so the only real version
; is on the text page, which would be
; destroyed if an evil hacker tried to
; break into the monitor right now)

```

68E4- 20 00 85   JSR  $8500

```

; and continue inside the new code

```

68E7- 4C C1 07   JMP  $07C1

```

Since \$8000+ is copied to \$0400+, \$07C1
is currently at \$83C1.

*83C1L

; I swear to God, it's another layer of
; encryption

```

83C1- A2 00      LDX  #$00
83C3- A0 03      LDY  #$03
83C5- BD C0 04   LDA  $04C0,X
83C8- 49 A5      EOR  $A5
83CA- 9D C0 04   STA  $04C0,X
83CD- E8         INX
83CE- D0 F5      BNE  $83C5
83D0- EE C7 07   INC  $07C7
83D3- EE CC 07   INC  $07CC
83D6- 88         DEY
83D7- D0 EC      BNE  $83C5
83D9- F0 01      BEQ  $83DC

```

I can reproduce this easily enough.
Rewriting it at \$0300:

```

0300- A2 00      LDX  #$00
0302- A0 03      LDY  #$03
0304- BD C0 80   LDA  $80C0,X
0307- 49 A5      EOR  $A5
0309- 9D C0 80   STA  $80C0,X
030C- E8         INX
030D- D0 F5      BNE  $0304
030F- EE 06 03   INC  $0306
0312- EE 0B 03   INC  $030B
0315- 88         DEY
0316- D0 EC      BNE  $0304
0318- 60         RTS

```

*300G

*BSAVE MDSADJ DECRYPTED THRICE,A\$6400,
L\$2400,S5,D1

I'm beginning to suspect that this disk
is nothing more than an infinite series
of decryption routines with a game
bolted on as an afterthought.

~

Chapter 10
In Which We Find Two Of Everything

Continuing from... um... \$83DC I guess
(branched from \$83D9):

*83DCL

```
83DC-  A9 01      LDA  #$01      ;A=01
83DE-  85 36      STA  $36        ; =01
83E0-  A9 01      LDA  #$01      ;A=01
83E2-  38         SEC
83E3-  69 00      ADC  #$00      ;A=02
83E5-  F0 0E      BEQ  $83F5      ;nope
83E7-  10 03      BPL  $83EC      ;yes

83E9-  30 F0      BMI  $83DB      ;fake
83EB-  3D 2A 85   AND  $852A,X    ;fake
83EE-  37         ???
```

*83ECL

```
83EC-  2A         ROL
83ED-  85 37      STA  $37        ; =04
```

Now (\$36) points to \$0401.

```
83EF-  D0 01      BNE  $83F2      ;yes
83F1-  6C 6C 36   JMP  ($366C)    ;fake
83F4-  00         BRK              ;fake
83F5-  4C 12 04   JMP  $0412      ;fake
83F8-  4C 00 04   JMP  $0400      ;fake
83FB-  4C 0F 06   JMP  $060F      ;fake
```

*83F2L

```
83F2-  6C 36 00   JMP  ($0036)
```

So execution continues at \$0401.

The code that ends up at \$0401 is currently in memory at \$8001, so I'm going to leave it there and try not to get too confused.

*8001L

```
8001-  A9 09      LDA  #$09
8003-  85 25      STA  $25
8005-  85 01      STA  $01
8007-  A9 00      LDA  #$00
8009-  85 24      STA  $24
800B-  85 83      STA  $83
800D-  85 00      STA  $00
800F-  20 C0 04   JSR  $04C0
```

That's at \$80C0.

*80C0L

```
80C0-  20 6F 85   JSR  $856F
```

That's actually at \$856F. Weird. This second RWTs relies on MDSADJ still being in memory. So many interlocking dependencies...

*856FL

```
856F-  A5 51      LDA  $51        ;A=00
8571-  49 AA      EOR  #$AA        ;A=AA
```

```
8573-  49 AA      EOR  #$AA        ;A=00
8575-  85 36      STA  $36        ; =00
8577-  49 1C      EOR  #$1C        ; =1C
8579-  85 37      STA  $37        ; =1C
857B-  60         RTS
```

Now (\$36) points to \$1C00.

Continuing from \$04C3, which is at \$80C3...

*80C3L

; looks like we're going to read that
; catalog sector again (T11,S07)

```
80C3-  A9 11      LDA  $11
80C5-  85 80      STA  $80
80C7-  A9 07      LDA  $07
80C9-  85 81      STA  $81
80CB-  85 82      STA  $82
80CD-  A5 D7      LDA  $D7
80CF-  8D 01 02   STA  $0201
80D2-  20 C3 06   JSR  $06C3
```

That's at \$82C3.

*82C3L

; (not shown) This is the sector read
; routine. It follows the same pattern
; as the first RWTs -- moving the drive
; head, reading a sector. But it has
; its own disk read routine, which
; explains why my copy hung with the
; drive motor on. It's actually trying
; to read T11,S07 and failing because
; it's looking for the protected data
; prologue ("D5 AA AA") again.

```
82C3-  20 FA 04   JSR  $04FA
```

The new sector read routine starts at \$0604, which is in memory at \$8204.

*8204L

```
8204-  18         CLC
8205-  08         PHP
8206-  BD 8C C0    LDA  $C08C,X
8209-  10 FB      BPL  $8206
820B-  49 D5      EOR  $D5
820D-  D0 F7      BNE  $8206
820F-  BD 8C C0    LDA  $C08C,X
8212-  10 FB      BPL  $820F
8214-  C9 AA      CMP  $AA
8216-  D0 F3      BNE  $820B
8218-  EA         NOP
8219-  BD 8C C0    LDA  $C08C,X
821C-  10 FB      BPL  $8219
821E-  C9 96      CMP  $96
8220-  F0 09      BEQ  $822B
8222-  28         PLP
8223-  90 DF      BCC  $8204
8225-  49 AA      EOR  $AA        <-- !
8227-  F0 25      BEQ  $824E
8229-  D0 D9      BNE  $8204
```

That \$AA at \$0626 needs to be changed to \$AD now that my disk uses standard data prologues. (This explains why my crack-in-progress hung with the drive motor on -- it was trying to read the disk with this second RWTs.)

Continuing from \$06C6 (after the call to \$04FA returns)...

*82C6L

```
82C6-  A6 2B      LDX    $2B
82C8-  20 EE 06   JSR    $06EE
82CB-  29 01      AND    #$01
82CD-  AA         TAX
82CE-  BD D4 06   LDA    $06D4,X
82D1-  F0 05      BEQ    $82D8
82D3-  60         RTS
82D4-  20 00 64   JSR    $6400
82D7-  4C A9 8F   JMP    $8FA9
```

Lots going on here. I'll look at \$06EE in a minute, but check out what we're doing with the return code. If the low bit of the accumulator is 0, X will end up being 0 and we'll read \$06D4 (=\$20), not take the "BEQ" branch, and exit via the "RTS" at \$06D3. But if the low bit of the accumulator is 1, X will end up being 1 and we'll read \$06D5 (=\$00), take the branch to \$06D8, and end up... doing what exactly?

*82D8L

```
82D8-  A9 8F      LDA    #$8F      ;A=8F
82DA-  18         CLC
82DB-  69 07      ADC    #$07      ;A=96
82DD-  85 36      STA    $36      ;=96
82DF-  A9 10      LDA    #$10      ;A=10
82E1-  38         SEC
82E2-  E9 0C      SBC    #$0C      ;A=04
82E4-  85 37      STA    $37      ;=04
```

Now (\$36) points to \$0496.

```
82E6-  2A         ROL          ;A=09
82E7-  2A         ROL          ;A=12
82E8-  85 39      STA    $39      ;=12
82EA-  6C 36 00   JMP    ($0036)
```

So we jump to \$0496, which is in memory at \$8096.

*8096L

```
8096-  A0 05      LDY    #$05
8098-  20 A2 04   JSR    $04A2
```

*80A2L

; wipe all memory, starting at the page ; given in A (=\$05)

```
80A2-  84 01      STY    $01
80A4-  A9 00      LDA    #$00
80A6-  85 00      STA    $00
80A8-  A8         TAY
80A9-  91 00      STA    ($00),Y
80AB-  C8         INY
80AC-  91 00      STA    ($00),Y
80AE-  C8         INY
80AF-  91 00      STA    ($00),Y
80B1-  C8         INY
80B2-  91 00      STA    ($00),Y
80B4-  C8         INY
80B5-  D0 F2      BNE    $80A9
80B7-  E6 01      INC    $01
80B9-  A6 01      LDX    $01
80BB-  E0 C0      CPX    #$C0
```

```
80BD-  D0 EA      BNE    $80A9
80BF-  60         RTS
```

So it appears that \$06EE is a Very Important Routine, and it is vital that the low bit of the accumulator end up being 0 at the end of it. It's the same logic as the protection check! Or should I say, the *first* protection check, because it appears that there are two of them.

Four decryption loop.
Three hidden files.
Two nibble checks.
And a classic game on a floppy.

Merry Crackmas.

~

Chapter 11
And A Happy New Year

Let's go look at \$06EE.

*82EEL

```
82EE-  BD 89 C0   LDA    $C089,X
82F1-  A9 56      LDA    #$56
82F3-  85 11      STA    $11
82F5-  D0 01      BNE    $82F8
82F7-  D0 C6      BNE    $82BF
82F9-  12         ???
```

If this were a job, I'd quit.

```
*82F7:EA
*82FE:EA
*8307:EA
*830F:EA
*8316:EA
*831E:EA
*8325:EA
*832D:EA
*8334:EA
*833E:EA
*8345:EA
*8348:EA
*8352:EA
*835A:EA
*8362:EA
*836A:EA
*8371:EA
*837A:EA
```

*82EEL

```
; turn on drive motor
82EE-  BD 89 C0   LDA    $C089,X
```

```
; initialize Death Counters
82F1-  A9 56      LDA    #$56
82F3-  85 11      STA    $11
82F5-  D0 01      BNE    $82F8
82F7-  EA         NOP
82F8-  C6 12      DEC    $12
82FA-  F0 03      BEQ    $82FF
82FC-  D0 0A      BNE    $8308
82FE-  EA         NOP
82FF-  C6 11      DEC    $11
8301-  D0 05      BNE    $8308
```

```
; if Death Counters hit 0, load A=FF
; and exit
8303- A9 FF      LDA    #$FF
8305- D0 7C      BNE    $8383
8307- EA        NOP
```

```
; find standard address prologue
; (D5 AA 96)
8308- BD 8C C0   LDA    $C08C,X
830B- 10 FB      BPL    $8308
830D- D0 01      BNE    $8310
830F- EA        NOP
8310- C9 D5      CMP    #$D5
8312- F0 03      BEQ    $8317
8314- D0 E2      BNE    $82F8
8316- EA        NOP
8317- BD 8C C0   LDA    $C08C,X
831A- 10 FB      BPL    $8317
831C- D0 01      BNE    $831F
831E- EA        NOP
831F- C9 AA      CMP    #$AA
8321- F0 03      BEQ    $8326
8323- D0 D3      BNE    $82F8
8325- EA        NOP
8326- BD 8C C0   LDA    $C08C,X
8329- 10 FB      BPL    $8326
832B- D0 01      BNE    $832E
832D- EA        NOP
832E- C9 96      CMP    #$96
8330- F0 03      BEQ    $8335
8332- D0 C4      BNE    $82F8
8334- EA        NOP
```

```
; skip over an $FF nibble
8335- A0 0A      LDY    #$0A
8337- BD 8C C0   LDA    $C08C,X
833A- 10 FB      BPL    $8337
833C- D0 01      BNE    $833F
833E- EA        NOP
833F- C9 FF      CMP    #$FF
8341- F0 03      BEQ    $8346
8343- D0 B3      BNE    $82F8
8345- EA        NOP
8346- F0 01      BEQ    $8349
8348- EA        NOP
```

```
; Read data latch exactly once (no BPL
; loop here!) and check its value.
; We're out of sync here because of all
; the branching, so the exact value of
; the data latch depends on timing bit
; after the $FF nibble. This is
; essentially the same technique as the
; first protection check, but done in a
; different way.
```

```
8349- BD 8C C0   LDA    $C08C,X
834C- C9 08      CMP    #$08
834E- B0 A8      BCS    $82F8
8350- D0 01      BNE    $8353
8352- EA        NOP
```

```
; calculate a checksum on the following
; nibbles
```

```
8353- BD 8C C0   LDA    $C08C,X
8356- 10 FB      BPL    $8353
8358- D0 01      BNE    $835B
835A- EA        NOP
835B- 85 10      STA    $10
835D- 88        DEY
835E- D0 03      BNE    $8363
8360- F0 10      BEQ    $8372
8362- EA        NOP
```

```
8363- BD 8C C0   LDA    $C08C,X
8366- 10 FB      BPL    $8363
8368- D0 01      BNE    $836B
836A- EA        NOP
836B- 45 10      EOR    $10
836D- D0 EC      BNE    $835B
836F- F0 EA      BEQ    $835B
8371- EA        NOP
```

```
; final checksum must be $60
8372- A5 10      LDA    $10
8374- 49 60      EOR    #$60
8376- D0 80      BNE    $82F8
8378- F0 01      BEQ    $837B
837A- EA        NOP
```

```
; wipe this entire protection check
; from memory
```

```
837B- A0 8E      LDY    #$8E
837D- 99 EE 06   STA    $06EE,Y
8380- 88        DEY
8381- D0 FA      BNE    $837D
```

```
; on exit, A=$00 on success (after
; falling through) or $FF on failure
; (coming from $0705)
8383- 60        RTS
```

Revisiting the caller at \$06C6...

*82C6L

```
; execute protection check
82C6- A6 2B      LDX    $2B
82C8- 20 EE 06   JSR    $06EE
```

```
; get low bit
82CB- 29 01      AND    #$01
```

```
; 0=success, 1=failure
```

```
82CD- AA        TAX
82CE- BD D4 06   LDA    $06D4,X
```

```
; failure path branches to The Badlands
82D1- F0 05      BEQ    $82D8
```

```
; success path returns to caller
82D3- 60        RTS
82D4- [20 00]
```

To bypass this, I can change the "AND #\$01" at \$06CB to "AND #\$00", to act as if the protection check always passes.

To sum up: MDSADJ decrypts itself three separate times, revealing a second RWTS and a second protection check. Now that I've decrypted it (three times), I can patch the second RWTS and disable the second protection check.

```
; jump to the start of the code that
; was decrypted by the first decryption
; loop (which has already been done)
*6400:4C 1D 64
```

```
; disable second decryption loop (also
; already done) by putting an "RTS" at
; the beginning of the routine at $8500
*8500:60
```

```
; skip over third decryption loop (also
```



Acknowledgements

*Great things in business are never done by one person.
They're done by a team of people. — Steve Jobs*

The Editor acknowledges the work of the following people in the production of this book:

DON WORTH AND PIETER LECHNER

Steven Weyhrich, on his website apple2history.org, posted an email from Don Worth where he provided the story of how the book came to be written.

Date: Wednesday, June 6, 2001 10:01 AM

Subject: Re: Beneath Apple DOS

In case you're interested, a friend and I disassembled DOS 3.1 (including RWTS – for which there was a listing in xeroxed form, and the File Manager). Vic Tolomei, who later became involved in writing books/software for the Exidy Sorcerer computer, prompted me to give it a try – I had been complaining about there being no documented way to open and read/write files from assembly language on the Apple II. Vic and I took apart my Disk II drive on his dining room table and twiddled the I/O addresses one by one to see what the arm did. Vic ran out a disassembly of DOS (using the disassembler in the Apple II ROM) and started marking it up – here and there, kind of like a crossword puzzle – I took what he started with and finished it off. When we were done, we had a fully commented listing of DOS.

Vic and I put our listing in a binder, comb bound it, and made a couple of copies for friends. I sent one copy up to

Apple just for fun. I got a somewhat panicked call from Woz. He wanted to know if we were planning to publish the source code – he was afraid it would damage Apple’s copyright or some such. I promised we would not. I got personal Christmas cards from him for the next few years.

I did some seminars at a computer store in Burbank, California on the API for the File Manager, which is where I met Pieter Lechner. We became friends, and a couple of years later decided to write *Beneath Apple DOS* together. Pieter wrote the low-level stuff about track formatting and RWTS. I wrote the higher level stuff about the File Manager. We included the comments from Vic’s and my source listing with address ranges – but I didn’t (technically) publish the source code. (A fine distinction – but by then I guess Woz no longer cared so much about it.) Since the File Manager API was still unknown, the book took off like a rocket. It’s interesting to consider that, if Shepardson had finished the documentation for DOS as Apple had wanted them to, then there probably wouldn’t have been a market for our book!

Later on we took apart Integer BASIC and the Galfo Integer BASIC runtime code so that we could port the latter to Atari’s and IBM PCs (which is how *Beneath Apple Manor* managed to move to those platforms) – but that’s another story. We also disassembled ProDOS when that came out – a much bigger job than DOS. I can’t even imagine trying to reverse engineer today’s OS’s – they are all produced from higher level language compilers anyway so they’d look pretty odd.

Don Worth has a personal website at <http://worth.bol.ucla.edu/> and is active on the Apple II Enthusiasts page.

SEAN MCNAMARA

Sean “europlus” McNamara is an Apple II enthusiast in Sydney, Australia. An Apple II user since 1982, he began collecting Apple IIs in 1998 when he was gifted a IIGs, with europluses being his passion. He acquired more Apple IIs over the years to prevent them being junked, and started being involved in the local community in 2011.

In 2015, he attended Oz KFest in Melbourne and really hit the ground running in the local community, including giving a presentation on solid state storage solutions for the Apple II. A week after his return from Melbourne he hosted the first of what would become WOzFests - one day gatherings of Apple II enthusiasts.

Now held three times a year, WOzFests see the introduction of new hardware and software products, Skypes to Apple II enthusiasts and retrocomputer luminaries worldwide, software and hardware preservation/repair, show-and-tells, chats, cider and pizza - an Apple II enthusiasts heaven!

4 AM*

It's tempting to rewrite history and give myself some noble purpose for starting this hobby, but in this case the truth makes for a better story. My parents bought themselves an Apple //e when I was 10, and it quickly came to dominate my leisure time. Pirated software was rampant, and I idolized the crackers whose names I saw flash and scroll on the "crack screens" of the games I traded with my friends. I also admired the few who documented their methods in cracking tutorials, initially distributed as BBS text files and later collated and redistributed on disk. I PEEK'd and POKE'd and CALL'd many late nights as a teenager, but I could never quite put it all together.

In late 2013, I acquired a real Apple //e and bought a few lots of original disks on eBay, mostly arcade games that I had acquired illicitly in my youth: *Sneakers*, *Repton*, *Dino Eggs*. To my surprise, the originals had more content than I remembered! *Sneakers* has an animated boot sequence. *Repton* has a multi-page introduction that explains the back story of the game. So I set out to create *complete* cracks that faithfully reproduced the original experience. I decided to document my methods because I enjoy technical writing, and because I had admired the classic crackers who had done so. (I decided to leave out the crack screens.)

One of those eBay lots had an educational game, *Ten Little Robots*. After cracking it, I couldn't find any copies of it online, which seemed odd. By 2014, surely everything has been cracked? Perhaps it was just misnamed or misfiled? Then I found another disk that seemed to be a first-time preservation. And another. And it slowly dawned on me that maybe not everything has been cracked.

I mentioned this to a friend of mine, and he set me straight. Preservation is driven by pirates, who are driven by ego but constrained by the technical limitations

* 4 am provided information for the ProDOS version of *Appendix B: Copy Protection*

of their era. In the 1980s, this meant storage space and modem speed. Nobody got kudos for cracking *Irregular Spanish Verbs in the Future Tense*, no BBS would waste the hard drive space to host it, and no user would sacrifice their phone line to download it. So it never got preserved in any form.

And even the things that did get cracked weren't fully preserved. Those same technical constraints led to a culture where the smallest version of a game always won. That meant stripping out the animated boot sequence, the title screen, the multi-page introduction, the cut scenes, anything deemed *non-essential* to the pirates. The *holy grail* was cutting away so much that you could distribute the game (or what was left of it) as a single file that could be combined with other unrelated games on a single floppy disk.

Thirty years later, that's exactly what I saw: half-preserved arcade games, a smattering of educational software, and virtually nothing else. I realized I could have a real impact while having just as much fun and just as much intellectual challenge. Along the way, I've discovered that educational software is rich with history, personality, humor, and technical achievement. It's been delightful.

Another early misunderstanding of mine was that copy protections were all unique. Nothing could be further from the truth! The most common protection schemes were the ones that were productized and resold to dozens of publishers. This was coordinated through the disk duplication houses, which offered copy protection as a *value add* on top of mastering the disks themselves. Publishers got the benefit of the latest and greatest copy protection without needing to play the cat-and-mouse game themselves. This led to some funny coincidences; my favorite is that *Math Blaster* and *Ultima IV* share the same copy protection. Two different publishers, two wildly different target audiences, but they had identical protection because protection was a product.

Of course, no one is inventing new Apple II protection schemes anymore. The cat-and-mouse game is over, and everyone lost. But make no mistake: copy protection works. I started cracking in earnest when I found one disk that had never been cracked. Since then, we've found thousands. With that many samples, we can start to look for patterns. Because most copy protection was an *add-on*, applied in bulk by automated processes, we can create similar processes to remove it in bulk.

And so we arrive at Passport. Passport is an automated disk verification and copy program. And when I say *automatic*, I mean it. Unlike classic nibble copiers, there are no parameters, no options, no knobs to fiddle. Also unlike classic copiers, the copy it produces is fully unprotected. No boot tracing, no





When you load too many ampersand routines into memory

sector editor patching on the back end. It's all built-in. Passport is a distillation of everything I've learned about cracking: every disk, every variation, every edge case.

This has completely changed my hobby. Passport ensures consistency. I don't worry about missing a patch or mistyping a hex value. I don't spend any time doing the grunt work that computers can do for me. If I find two disks with the same protection, I write a new Passport module to automate it. The same module that cracks *Ultima IV* also cracks *Math Blaster*. Remember, protection was productized. If I've found two samples of a protection scheme, there are likely twenty more. They're out there, undiscovered, rotting away on physical media.

Automation frees me to look beyond the bits. I can spend more time on in-depth write-ups of protection schemes that can't be automated. I can take screenshots and make boot videos to show off all the wonderful educational software I've discovered. The copy protection is the least interesting part of these disks. It's just the part that prevented us from studying all the other parts.

Passport is online at <https://archive.org/details/Passport4am>

Glossary

access time. The time required to locate and read or write data on a direct access storage device, such as a diskette drive.

address. The numeric location of a piece of data in memory, usually given as a hexadecimal number from \$0000 to \$FFFF (65,535 decimal). A disk address is the location of a data sector expressed in terms of its track and sector numbers.

algorithm. A sequence of steps which may be performed by a program or other process, which will produce a given result.

alphanumeric. An alphabetic character (A-Z) or a numeric digit (0-9). In the past, the term referred to the class of all characters and digits.

analog. Having a value which is continuous, such as a voltage or electrical resistance, as opposed to digital.

AND. The logical process of determining whether two bits are both ones. 0 AND 1 results in 0 (false), 1 AND 1 results in 1 (true).

arm. The portion of a disk drive which suspends the read/write head over the disk's surface. The arm can be moved radially to allow access to different tracks.

ASCII. American Standard Code for Information Interchange. A hexadecimal to character conversion code assignment, such that the 256 possible values of a single byte may each represent a alphabetic, numeric, special, or control character. ASCII is used when interfacing to peripherals, such as keyboards, printers, or video text displays.

assembly language. Also known as machine language. The native programming language of the individual computer. Assembly language is oriented to the machine, and is not humanized as is BASIC, PASCAL, or FORTRAN. An assembler is used to convert assembly language statements to an executable program.

backup. The process of making a copy of a program or data against the possibility of its accidental loss or destruction.

bank switched memory. Also called the language card. An additional 16K of memory which may only be accessed by "throwing" hardware switches to cause portions of the bank switched memory to temporarily replace the Monitor ROM memory in the machine. This is necessary because an Apple can only address 64K, and all addresses are already used with 48K, 4K of I/O and 12K of Monitor ROM.

base. The number system in use. Decimal is base 10, since each digit represents a power of 10 (1,10,100,...). Hexadecimal is base 16 (1,16,256,...). Binary is base 2 (1,2,4,8,...).

BI (BASIC Interpreter). Also called the BASIC System Program. The BI accepts user commands such as CATALOG and LOAD, and translates them into calls to the ProDOS Machine Language Interface (MLI).

binary. A number system based upon powers of 2. Only the digits 0 and 1 are used. For example, 101 in binary is 1 units digit, 0 twos, and 1 fours, or 5 in decimal.

bit cell. The space on a diskette which passes beneath the read/write head in a 4-μsecond interval. A bit cell contains a signal which represents the value of a single binary 0 or 1 (bit).

bit map. A table where each binary bit represents the allocation of a unit of storage. ProDOS uses bit maps to keep track of memory use (System Bit Map) and of disk use (Volume Bit Map).

bit slip marks. The epilogue of a disk field, used to double check that the disk head is still in read sync and the sector has not been damaged.

bit. A single binary digit (a 1 or a 0). A bit is the smallest unit of storage or information in a computer.

block. An arbitrary unit of disk space composed of two sectors or 512 bytes. ProDOS reads and writes a block at a time to improve performance and to allow support for larger devices.

boot/bootstrap. The process of loading a very large program into memory by loading successively larger pieces, each of which loads its successor. The program loads itself by “pulling itself up by its bootstraps”.

BRK/Break. An assembly language instruction which can be used to force an interrupt and immediate suspension of execution of a program.

buffer. An area of memory used to temporarily hold data as it is being transferred to or from a peripheral, such as a disk drive.

bug. A programming error. Faulty operation of a program.

byte. The smallest unit of addressable memory in a computer. A byte usually consists of 8 bits and can contain a decimal number ranging from 0 to 255 or a single alphanumeric character.

carriage return. A control character which instructs the printer to end one line and begin another. When printing a carriage return is usually followed by a line feed.

carry flag. A 6502 processor flag which indicates that a previous addition resulted in a carry. Also, used as an error indicator by many system programs.

catalog. A directory of the files on a diskette. See directory.

chain. A linked list of data elements. Data is chained if its elements need not be contiguous in storage and each element can be found from its predecessor via an address or block pointer.

checksum/CRC. A method for verifying that data has not been damaged. When data is written, the sum of all its constituent bytes is stored with it. If, when the data is later read, its sum no longer matches the checksum, it has been damaged.

clobbered. Damaged or destroyed. A clobbered sector is one which has been overwritten such that it is unrecoverable.

code. Executable instructions to the computer, usually in machine language.

coldstart. A restart of a program which reinitializes all of its parameters, usually erasing any work which was in progress at the time of the restart.

A DOS coldstart erases the BASIC program in memory.

contiguous. Physically next to. Two bytes are contiguous if they are adjoining each other in memory or on disk.

control block. A collection of data which is used by the operating system to manage resources. Examples of a control block used by DOS are the file buffers. Examples of control blocks used by ProDOS are the Volume Control Block (VCB) or a Volume Directory Header.

control character. A special ASCII code which is used to perform a unique function on a peripheral, but does not generate a printable character. Carriage return, line feed, form feed, and a bell are all control characters.

controller card. A hardware circuit board which is plugged into an Apple connector which allows communication with a peripheral device, such as a disk or printer. A controller card usually contains a small driver program in ROM.

CSWL. A vector in zero page, through which output data is passed for display on the CRT or for printing.

cycle. The smallest unit of time within the central processor of the computer. Each machine language instruction requires two or more cycles to complete. One cycle on the Apple is about one microsecond (one millionth of a second).

data sector buffer. On the Apple, a 256 byte buffer used by DOS to hold the image of any given sector on the diskette. As information is read from the file, data is extracted from the data sector buffer until it is exhausted, at which time it is refilled with the next sector image.

data type. The type of information stored in a byte. A byte might contain a printable ASCII character, binary numeric data, or a machine language instruction.

data. Units of information.

DCT. Device Characteristics Table. Used as an input parameter table to Read/Write Track/Sector (RWTS) to describe the hardware characteristics of the diskette drive.

decimal. A number system based upon powers of 10. Digits range from 0 to 9.

deferred commands. DOS commands which may (or must) be invoked from within an executing

- BASIC program. OPEN, READ, WRITE, and CLOSE are all examples of deferred commands. ProDOS commands which may (or must) be invoked from within an executing BASIC program. OPEN, APPEND, READ, WRITE, and CLOSE are all examples of deferred commands.
- digital.** As opposed to analog. Discrete values as opposed to continuous ones. Only digital values may be stored in a computer. Analog measurements from the real world, such as a voltage or the level of light outside, must be converted into a numerical value which, of necessity, must be “rounded off” to a discrete value.
- direct access.** Peripheral storage allowing rapid access of any piece of data, regardless of its placement on the medium. Magnetic tape is generally not considered direct access, since the entire tape must be read to locate the last byte. A diskette is considered direct access, since the arm may be rapidly moved to any track and sector.
- directory.** A catalog of files stored on a diskette. The directory must contain each file’s name and its location on the disk as well as other information regarding the type of data stored there. In ProDOS, a directory is a file in itself and one directory can describe other subdirectories.
- disk initialization.** The process which places track formatting information, including sectors and gaps, on a blank diskette. During disk initialization, DOS also places a VTOC and directory on the newly formatted disk, as well as saving the HELLO program, and the ProDOS FILER also places a copy of the boot loader in Block 0 and creates an empty Volume Directory in Blocks.
- displacement.** The distance from the beginning of a block of data to a particular byte or field. Displacements are usually given beginning with 0, for the first byte, 1 for the second, etc. Also known as an offset.
- DOS.** Also called DOS 3.2 and DOS 3.3. An earlier disk operating system for the Apple. DOS was designed to support BASIC programming using the Disk II drive only. When hard disks became available, Apple introduced ProDOS.
- driver.** A program which provides an input stream to another program or an output device. A printer driver accepts input from a user program in the form of lines to be printed, and sends them to the printer.
- dump.** An unformatted or partially formatted listing of the contents of memory or a diskette in hexadecimal. Used for diagnostic purposes.
- encode.** To translate data from one form to another for any of a number of reasons. In DOS 3.3 and ProDOS, data is encoded from 8 bit bytes to 6 bit bytes for storage on a Disk II.
- entry point (EPA).** The entry point address is the location within a program where execution is to start. This is not necessarily the same as the load point (or lowest memory address of the program).
- EOF (End Of File).** This mark signals the end of a data file. \$00 for APPLE DOS text files. For ProDOS, a 3-byte number ranging from 0 to 16,777,216 (16 megabytes), which represents the offset to the end of the file. If the file is sequential (contains no “holes”), the EOF is also the length of the file in bytes.
- epilogue.** The last three bytes of a field on a track. These unique bytes are used to insure the integrity of the data which precedes them.
- Exclusive OR.** A logical operation which compares two bits to determine if they are different. 1 EOR 0 results in 1. 1 EOR 1 results in 0.
- field.** A group of contiguous bytes forming a single piece of data, such as a person’s name, age, or social security number. In disk formatting, a group of bytes surrounded by gaps.
- file buffers.** In Apple DOS, a collection of buffers used to manage one open file. Included are a data sector buffer, a Track/Sector List sector buffer, a file manager workarea buffer, the name of the file, and pointers. The DOS command, MAXFILES 3, causes 3 of these file buffers to be allocated. In Apple ProDOS, a pair of 512-byte buffers used by the BASIC interpreter to manage one open file. Included are a buffer containing the block image of the current index block and one containing the image of the current data block. File buffers are allocated by the BI as needed by moving Applesoft’s HIMEM pointer down in memory.

file descriptive entry. A single entry in a disk directory which describes one file. Included are the name of the file, its data type, its length, its access restrictions, its creation date, its location on the diskette, etc.

file manager. That portion of DOS which manages files. The file manager handles such general operations as OPEN, CLOSE, READ, WRITE, POSITION, RENAME, DELETE, etc.

file type. The type of data held by a file. Valid DOS file types are Binary, Applesoft, Integer-BASIC, Text, Relocatable, S, A, and B. Valid ProDOS file types include Binary (BIN), Applesoft (BAS), Text (TXT), and System (SYS) files. ProDOS supports up to 256 different file types.

file. A named collection of data on a diskette or other mass storage medium. Files can contain data or programs.

firmware. A middle ground between hardware and software. Usually used to describe micro-code or programs which have been stored in read-only memory (ROM).

gaps. The spaces between fields of data on a diskette. Gaps on an Apple diskette contain self-sync bytes.

garbage collection. The process of combining many small embedded free spaces into one large area. For example, Applesoft performs garbage collection on its string storage to recover memory allocated to strings which have been deleted.

Global Page. A 256-byte area of memory set aside by ProDOS to contain system variables of general interest. Two Global Pages are currently defined: the System Global Page at \$BF00; and the BI Global Page at \$BE00. The structure of the Global Pages is rigidly defined, allowing external programs to communicate with ProDOS without depending upon release dependent locations. See also vectors.

hard error. An unrecoverable Input/Output error. The data stored in the disk sector can never be successfully read again.

hardware. Physical computer equipment, as opposed to programs which run on the equipment. A disk drive is an example of a hardware component.

head. The read/write head on a diskette drive. A magnetic pickup, similar in nature to the head on

a stereo tape deck, which rests on the spinning surface of the diskette.

hexadecimal/HEX. A numeric system based on powers of 16. Valid hex digits range from 0 to 9 and A to F, where A is 10, B is 11, . . . F is 15. Standard Apple practice is to indicate a number as hexadecimal by preceding it with a dollar sign. \$B30 is 11-256s plus 3-16s plus 0-1s, or 2864 in decimal. Two hexadecimal digits can be used to represent the contents of one byte. Hexadecimal is used with computers because it easily converts to binary.

high memory. The memory locations which have high address values. \$FFFF is the highest memory location. Also called the “top” of memory.

HIMEM. Applesoft’s zero-page address which identifies the first byte past the available memory which can be used to store BASIC programs and their variables.

I/O (Input/Output) error. An error which occurs during transmission of data to or from a peripheral device, such as a disk or cassette tape.

immediate command. A DOS or ProDOS command which may be entered at any time, especially when the operating system is waiting for a command from the keyboard. Deferred commands are the opposite of immediate commands.

index block. A block containing a table of block numbers describing the order and location of the blocks of data within a file. A sapling file has one index block describing up to 256 data blocks. A tree file has a master index block which points to other index blocks, which in turn point to the data blocks in the file.

index. A displacement into a table or block of storage.

instruction. A single step to be performed in an assembly language or machine language program. Instructions perform such operations as addition, subtraction, store, or load.

integer. A “whole” number with no fraction associated with it, as opposed to floating point.

intercept. A program which logically places itself in the execution path of another program, or pair of programs. A video intercept is used to redirect program output from the screen to a printer, for example.

- interleave.** The practice of selecting the order of sectors on a diskette track to minimize access time due to rotational delay. Also called “skewing” or interlacing.
- interpreter.** A program which translates user written commands or program statements directly into their intended function. Applesoft is an interpreter. The ProDOS BASIC Interpreter translates ProDOS commands into functions such as loading, saving, reading or writing files. Another name for ProDOS interpreters is System Programs.
- interrupt.** A hardware signal which causes the computer to halt execution of a program and enter a special handler routine. Interrupts are used to service real-time clock time-outs, BRK instructions, and RESET.
- IOB.** DOS Input/Output Block. A collection of parameter data, passed to Read/Write Track/Sector, describing the operation to be performed.
- JMP.** A 6502 assembly language instruction which causes the computer to begin executing instructions at a different location in memory. Similar to a GOTO statement in BASIC.
- JSR.** A 6502 assembly language instruction which causes the computer to “call” a subroutine. Similar to a GOSUB statement in BASIC.
- K.** A unit of measurement, usually applied to bytes. 1K bytes is equivalent to 1024 bytes.
- Kernel.** That part of ProDOS which provides the basic operating system support functions. The Kernel resides in the Language Card or bank switched memory and consists of the MLI, interrupt handler, and diskette and calender/clock device drivers.
- key block.** The first block of a ProDOS file.
- KSWL.** A vector in zero page through which input data is passed from the keyboard or a remote terminal.
- label.** A name associated with a location in a program or in memory. Labels are used in assembly language much like statement numbers are used in BASIC.
- language card.** An additional 16K of RAM added to an Apple II or Apple II Plus using a card in slot 0. The card gets its name from its original use with the Apple UCSD PASCAL system and for loading other versions of BASIC. Apple IIe’s have this additional memory built in. See also bank switched memory.
- latch.** A component into which the Input/Output hardware can store a byte value, which will hold that value until the central processor has time to read it (or vice versa).
- link.** An address or block pointer in an element of a linked chain of data or buffers.
- list.** A one dimensional sequential array of data items.
- load point (LP).** The lowest address of a loaded assembly language program—the first byte loaded. Not necessarily the same as the entry point address (EPA).
- locked.** A file is locked if it is restricted from certain types of access—usually one which is read only. ProDOS provides control over file access through the use of directory entry bits.
- logical.** A form of arithmetic which operates with binary “truth” or “false”, 1 or 0. AND, OR, NAND, NOR, and EXCLUSIVE OR are all logical operations.
- LOMEM.** Applesoft’s zero-page address which identifies the first byte of available memory which can be used to store BASIC programs and their variables.
- loop.** A programming construction in which a group of instructions or statements are repeatedly executed.
- low memory.** The memory locations with the lowest addresses. \$0000 is the lowest memory location. Also called the “bottom” of memory.
- LSB/lo order.** Least Significant Bit or Least Significant Byte. The 1’s bit in a byte or the second pair of hexadecimal digits forming an address. In the address \$8030, \$30 is the LO order part of the address.
- mark.** A 3-byte “byte number” or position within a ProDOS file. When a file is being read by the MLI, a current mark is maintained as well as the EOF mark. See also EOF.
- master disk.** A DOS diskette which will boot in an Apple II of any size memory and take full advantage of it.
- microsecond.** A millionth of a second. Equivalent to one cycle of the Apple II central processor. Also written as “μsec”

MLI (Machine Language Interface). The MLI is part of the ProDOS Kernel which resides in the language card or bank switched memory. The MLI performs such functions as OPENing a file, WRITing to a file, or DESTROYing a file.

monitor. A machine language program which always resides in the computer and which is the first to receive control when the machine is powered up. The Apple monitor resides in ROM and allows examination and modification of memory at a byte level.

MSB/hi order. Most Significant Bit or Most Significant Byte. The 128's bit of a byte (the left-most) or the first pair of hexadecimal digits in an address. In the byte value \$83, the MSB is on (is a 1).

nibble/nybble. A portion of a byte, usually 4 bits and represented by a single hexadecimal digit. \$FE contains two nibbles, \$F and \$E.

null. Empty, having no length or value. A null string is one which contains no characters. The null control character (\$00) produces no effect on a printer (also called an idle).

object code. A machine language program in binary form, ready to execute. Object code is the output of an assembler.

object module. A complete machine language program in object code form, stored as a file on a diskette.

offset. The distance from the beginning of a block of data to a particular byte or field. Offsets are usually given beginning with 0, for the first byte, 1 for the second, etc. Also known as a displacement.

opcode, operation code. The three letter mnemonic representing a single assembly language instruction. JMP is the opcode for the jump instruction.

operating system. A machine language program which manages the memory and peripherals automatically, simplifying the job of the applications programmer.

OR. The logical operation comparing two bits to determine if either of them are 1. 1 OR 1 results in 1 (true). 1 OR 0 results in 1, 0 OR 0 results in 0 (false).

overhead. The space required by the system, either in memory or on the disk, to manage either. In DOS, the disk directory and VTOC are part of a

diskette's overhead. In ProDOS, the boot blocks, Volume Directory, and Volume Bit Map are part of a diskette's overhead.

page. 256 bytes of memory which share a common high order address byte. Zero page is the first 256 bytes of memory (\$0000 through \$00FF).

parallel. A communication mode which sends all bits in a byte at once, each over a separate line or wire. Opposite of serial.

parameter list. An area of storage set aside for communication between a calling program and a subroutine. The parameter list contains input and output variables which will be used by the subroutine.

parity. A scheme which allows detection of errors in a single data byte, similar to checksums but on a bit level rather than a byte level. An extra parity bit is attached to each byte which is a sum of the bits in the byte. Parity is used to detect or correct single bit failures, and when sending data over communication lines to detect noise errors.

parse. The process of interpreting character string data, such as a command with keywords.

patch. A small change to the object code of an assembly language program. Also called a "zap".

pathname. A string describing the path ProDOS must follow to find a file. A fully qualified pathname consists of the volume name followed by one or more directory names followed by the name of the file itself. If a partial pathname is given, a default prefix is attached to it to form a complete pathname. See also prefix.

peripheral. A device which is external to the computer itself, such as a disk drive or a printer. Also called an Input/Output device.

physical record. A collection of data corresponding to the smallest unit of storage on a peripheral device. For disks, a physical record is a sector.

pointer. The address or memory location of a block of data or a single data item. The address "points" to the data. A pointer may also be a block number, such as the pointer to the Volume Bit Map in the Volume Directory Header.

prefix. A system maintained default character string which is automatically attached to file names entered by the user to form a complete pathname. See also pathname.

prologue. The three bytes at the beginning of a disk field which uniquely identify it from any other data on the track.

PROM (Programmable Read Only Memory).

PROMs are usually used on controller cards associated with peripherals to hold the driver program which interfaces the device to applications programs.

prompt. An output string which lets the user know that input is expected. An “*” is the prompt character for the Apple monitor.

protected disk. A diskette whose format or content has been modified to prevent its being copied. Most retail software today is distributed on protected disks to prevent theft.

pseudo-opcode. A special assembly language opcode which does not translate into a machine instruction. A pseudo-opcode instructs the assembler to perform some function, such as skipping a page in an assembly listing or reserving data space in the output object code.

RAM (Random Access Memory). Computer memory which will allow storage and retrieval of values by address.

random access. Direct access. The capability to rapidly access any single piece of data on a storage medium without having to sequentially read all of its predecessors.

recal. Recalibrate the disk arm so that the read/write head is positioned over track zero. This is done by pulling the arm as far as it will go to the outside of the diskette until it hits a stop, producing a “clacking” sound.

record. A collection of associated data items or fields. One or more records are usually associated with a file. Each record might correspond to an employee, for example.

reference number (REF_NUM). An arbitrary number assigned to an open file by the MLI to simplify identification in later calls.

register. A named temporary storage location in the central processor itself. The 6502 has 5 registers: the A, X, Y, S, and P registers. Registers are used by an assembly language program to access memory and perform arithmetic.

release. A version of a distributed piece of software. There have been several releases of DOS.

relocatable. The attribute of an object module file which contains a machine language program and the information necessary to make it run at any memory locations.

return code. A numeric value returned from a subroutine, indicating the success or failure of the operation attempted. A return code of zero usually means there were no errors. Any other value indicates the nature of the error, as defined by the design of the subroutine.

ROM (Read Only Memory). Memory which has a permanent value. The Apple monitor and Applesoft BASIC are stored in ROM.

RWTS (Read/Write Track/Sector). A collection of subroutines which allow access to the diskette at a track and sector level. RWTS is part of DOS and may be called by external assembly language programs.

sapling. A ProDOS file which requires only one index block (2 to 256 data blocks). A sapling ranges from 513 bytes to 131,072 bytes in length. See also seedling and tree.

search. The process of scanning a track for a given sector.

sector address. A disk field which identifies the sector data field which follows in terms of its volume, track, and sector number.

sector data. A disk field which contains the actual sector data in nibblized form.

sector. The smallest updatable unit of data on a disk track. One sector on an Apple Disk II contains 256 data bytes.

seedling. A ProDOS file which has only a single data block (512 bytes). A seedling file does not require index blocks. See also sapling and tree.

seek. The process of moving the disk arm to a given track.

self-sync. Also called “auto-sync” bytes. Special disk bytes which contain more than 8 bits, allowing synchronization of the hardware to byte boundaries when reading.

sequential access. A mode of data retrieval where each byte of data is read in the order in which it was written to the disk.

serial. A communication mode which sends data bits one at a time over a single line or wire. As opposed to parallel.

shift. A logical operation which moves the bits of a byte either left or right one position, moving a 0 into the bit at the other end.

skewing. The process of interleaving sectors. See *interleave*.

slave disk. A diskette with a copy of DOS which is not relocatable. The DOS image will always be loaded into the same memory location, regardless of the size of the machine.

soft error. A recoverable I/O error. A worn diskette might produce soft errors occasionally.

software. Computer programs and data which can be loaded into RAM memory and executed.

SOS (Sophisticated Operating System). The standard operating system for the Apple III computer.

source code. A program in a form which is understandable to humans; in a character form as opposed to internal binary machine format. Source assembly code must be processed by an assembler to translate it into machine or “object” code.

sparse file. A files with random organization (see *random access*) which contains areas which were never initialized. A sparse file might have an `END_OF_FILE` mark of 26 megabytes but only contain several hundred bytes.

state machine. A process (in software or hardware) which defines a unique target state, given an input state and certain conditions. A state machine approach is used in the ProDOS BASIC Interpreter to keep track of its video intercepts and by the hardware on the disk controller card to process disk data.

strobe. The act of triggering an I/O function by momentarily referencing a special I/O address. Strobing `$C030` produces a click on the speaker. Also called “*toggling*”.

subroutine. A program whose function is required repeatedly during execution, and therefore is called by a main program in several places.

system disk. A ProDOS volume which contains the system files necessary to allow ProDOS to be booted into memory. Normally, the PRODOS and BASIC.SYSTEM files are necessary. A STARTUP program may also be present.

system program. A ProDOS program, written in machine language, which acts as an intermediary

between the user and the ProDOS Kernel. BASIC.SYSTEM, FILER, and CONVERT are all examples of System Programs. See also *interpreter* and *BI*.

T/S list. Track/Sector List. A sector which describes the location of a file by listing the track and sector number for each of its data sectors in the order that they are to be read or written.

table. A collection of data entries, having a similar format, residing in memory. Each entry might contain the name of a program and its address. For examples. A “lookup” can be performed on such a table to locate any given program by name.

toggle. The act of triggering an I/O function by momentarily referencing a special I/O address. Toggling `$C030` produces a click on the speaker. Also called “*strobe*”.

tokens. A method where human recognizable words may be coded to single binary byte values for memory compression and faster processing. BASIC statements are tokenized, where hex codes are assigned to words like IF, PRINT, and END.

track. One complete circular path of magnetic storage on a diskette. There are 35 concentric tracks on an Apple diskette.

translate table. A table of single byte codes which are to replace input codes on a one-for-one basis. A translate table is used to convert from 6-bit codes to disk codes.

tree. A ProDOS file which requires several index blocks (131,073 to 16,777,216 bytes of data). See also *index block*, *seedling*, and *sapling*.

TTL (Transistor to Transistor Logic). A standard for the interconnection of integrated circuits which also defines the voltages which represent 0s and 1s.

unlocked. A file which allows all types of access (READ, WRITE, DELETE, RENAME, etc.). See also *locked*.

utility. A program which is used to maintain, or assist in the development of, other programs or disk files.

vector. A collection of pointers or JMP instructions at a fixed location in memory which allow access to a relocatable program or data.

Volume Directory. The first directory on a disk volume. Also called the “*root*” directory. All

other directories must be reached by first reading the Volume Directory.

volume. An identification for a diskette, disk platter, or cassette, containing one or more files.

VTOC. Volume Table Of Contents. Based upon the IBM OS/VS VTOC. On the APPLE, a sector mapping the free sectors on the diskette and giving the location of the directory.

warmstart. A restart of a program which retains, as much as is possible, the work which was in progress at the time. A DOS warmstart retains the BASIC program in memory.

write protected. A diskette whose write protect notch is covered, preventing the disk drive from writing on it.

ZAP. From the IBM mainframe utility SUPERZAP. A program which allows updates to a disk at a byte level, using hexadecimal.

zero page. The first 256 bytes of memory in a 6502 based machine. Zero page locations have special significance to the central processor, making their management and assignment critical.

ProDOS

DOS 3.3



Pascal

CP/M



ONLINE LINKS FOR THE APPLE II COMMUNITY

Call A.P.P.L.E., an Apple user group

<https://www.callapple.org/>

applefritter, an online forum

<https://www.applefritter.com/>

A2Central

<https://a2central.com/>

Apple 2 Online, a resource for Apple II documentation

<https://apple2online.com/>

Apple II Enthusiasts Group on Facebook

<https://www.facebook.com/groups/5251478676>

Apple II History, the definitive history of the Apple II computer

<https://apple2history.org/>

WOzFest

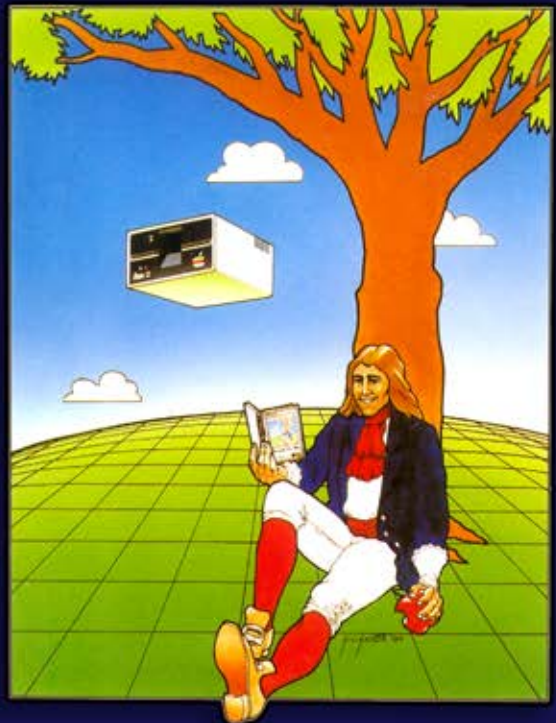
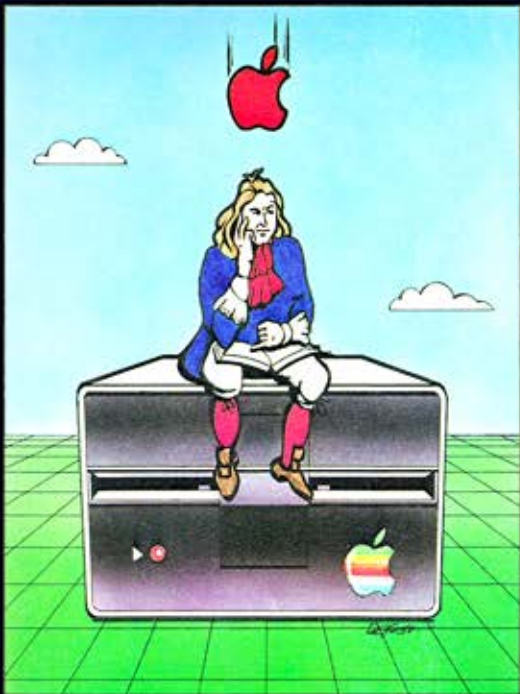
<https://the.europlus.zone/WOzFest>

Oz KFest

<http://ozkfest.net>

Apple Iloz Mailing List

http://www.appleiioz.com/appleiioz/Apple_Iloz/Mailing_List.html



From the Editor:

This book combines two works from the 1980s: *Beneath Apple DOS* and *Beneath Apple ProDOS*. I have tried to keep as much of the original text as possible, but some of the figures and tables have been re-drawn to either make them easier to read or easier to understand (or both).

Also, some of the information is duplicated in both books, but I have kept the duplicates so if you're studying one operating system, you don't need to reference the other. The one main item I combined were the glossaries, since many of the same terms are in both. It is the last chapter in the book.

If this is your first time seeing this book, I hope you experience the joy so many have over the years. If you had this book previously, I hope I've kept the feel of the original book, while updating it for a new generation.

— John Snape